AN ABSTRACT OF THE THESIS OF

Mihai Codoban for the degree of Master of Science in Computer Science presented on June 17, 2015.

Title: A Comparative Study on How SVN and Git Affect Software Changes

Abstract approved: _____

Daniel Dig

Distributed Version Control Systems (DVCS) have seen an increase in popularity relative to traditional Centralized Version Control Systems (CVCS). Yet we know little on whether VCS tools meet the needs of software developers when managing software change or whether developers are benefitting from the extra power of DVCS. Without such knowledge, researchers, developers, tool builders, and team managers are in the danger of making wrong assumptions.

In this paper we present the first in-depth, large scale empirical study that looks at the influence of DVCS on the practice of splitting, grouping, and committing changes.

We recruited 820 participants for a survey that sheds light into the practice of using DVCS and interviewed 13 participants on the practice of managing and communicating software changes. We also analyzed 409M lines of code changed by 358300 commits, made by 5890 developers, in 132 repositories containing a total of 73M LOC. Using this data, we uncovered some interesting facts. For example, (i) commits made in distributed repositories were 32% smaller than the centralized ones, (ii) developers split commits more often in DVCS, (iii) DVCS commits are more likely to have references to issue tracking labels, and (iv) developers use both ad-hoc and structured methods in communicating finished changes.

A Comparative Study on How SVN and Git Affect Software Changes

by

Mihai Codoban

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Science

Presented June 17, 2015
Commencement June 2016

Master of Science thesis of Mihai Codoban presented on June 17, 2015.

APPROVED:

Major Professor, representing Computer Science

Director of the School of Electrical Engineering and Computer Science

---

Dean of the Graduate School

I understand that my thesis will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my thesis to any reader upon request.

Mihai Codoban, Author

ACKNOWLEDGEMENTS

I would like to thank my advisor, Danny Dig, for making this research possible and for mentoring me as a graduate student.

I express sincere gratitude to my coauthors, Caius Brindescu and Sergii Shmarkatiuk, without whom this work would not have been an ICSE paper. It has been a great experience working with you.

I thank my committee members, Margaret Burnett, Carlos Jensen, Ron Metoyer, and Karl Schilke for taking the time to read this thesis and attend the defense.

I also thank Nicole Thompson for guiding me through the defense process.

Publications are not possible without tremendous amounts of feedback. I thank Cosmin Rădoi, Alexandru Gyori, David Hartveld, Alex Groce, Michael Rosulek, Will Jernigan, Faezah Bahmani, Iftakar Ahmed, Michael Hilton, Irwin Kwan, Charles Hill, Amber Horvath, Paul McKenney, Michael Slater, Rahul Gopinath, Alden Snow, and the anonymous reviewers for feedback on earlier drafts of this paper.

Also, I would like to thank Joel Spolsky, Robert Martin, and Steve Berczuk for helping us promote the survey.

This research is partly funded through NSFCCF-1439957 and CCF-1442157 grants, a SEIF award from Microsoft, and a gift grant from Intel.

Last but not least, I am eternally grateful to my parents and my wife, who have supported me throughout my graduate program and my life.

CONTRIBUTION OF AUTHORS

Caius Brindescu and I worked on this project from September 2012 to September 2013 and Sergii Shmarkatiuk joined us in the summer of 2013.

The repository analyzer was built mainly by Caius and I. Sergii contributed analyses regarding branching and issue tracking label detection.

I was the main author responsible for gathering and analyzing the repository corpus. Caius and Sergii also contributed to repository analysis.

The three of us designed and piloted the survey, Sergii and Caius coded the open ended questions, and Caius mainly analyzed the responses.

I designed, piloted, ran, and analyzed the interviews. Caius helped with qualitative coding.

While all three of us contributed to the ICSE paper, Caius was the master LaTeX typesetter.

Danny Dig advised and mentored us throughout the entire project. His assistance was instrumental to the project.

TABLE OF CONTENTS

Page

1 Introduction 1

2 Terminology 6
2.1 The centralized and the distributed model 6
2.2 Is a Git commit equivalent to an SVN commit? 9
2.3 Measuring commit size 10

3 Experimental Setup 11
3.1 Survey 11
3.2 Developer Interviews 13
3.3 Repository 14
3.3.1 Repository Corpus 14
3.3.2 Repository Analysis 17

4 Results 21
4.1 How does the type of VCS affect developers' behavior? 21
4.1.1 Implications 32
4.2 How does the type of VCS affect the development process? 36
4.2.1 Implications 42

5 Threats to validity 44

6 Related Work 47

7 Conclusions 49

Bibliography 50

Appendices 56
A Survey and interview questions 57
B Qualitative codes 59

LIST OF FIGURES

Figure		Page
2.1	The centralized and distributed paradigms	7
2.2	An example of a commit history in DVCS.	9
3.1	Tool stack	18

LIST OF TABLES

Table		Page
3.1	Mapping of data sources to RQs	11
3.2	Demographics of survey respondents	12
3.3	Demographics of interviewed participants.	14
3.4	Repository corpus.	16
4.1	Commit size across different VCS in terms of source lines changed (LOCC), and files changed (FC)	21
4.2	Developers splitting their commits (%)	23
4.3	Reasons for splitting commits (%)	24
4.4	Developers squashing their commits (%)	28
4.5	Reasons why developers squash their commits (%)	28
4.6	Reasons for considering a VCS more "natural" to commit (%)	29
4.7	How often do developers commit? (%)	31
4.8	Survey: How do developers commit if they work on more than one issue (%)	36
4.9	Average LOCC for issue references by VCS type	37
4.10	Correlation between commit size and commit time	38

LIST OF APPENDIX TABLES

Table		Page
A.1	Interview Questions.	57
A.2	Survey questions. Open-ended answers are marked with an asterisk.	58
B.1	Codes extracted from interview questions Q2 and Q4	59
B.2	Codes extracted from the survey open ended question Q10	60
B.3	Additional Codes that refer to splitting commits extracted from interview questions Q1 and Q2	60
B.4	Codes extracted from the survey open ended question Q12	61
B.5	Codes extracted from the survey open ended question Q14	61

Chapter 1: Introduction

Distributed Version Control Systems (DVCS) like Git [3] or Mercurial [9] are widely used today. GitHub [5], which is the most popular repository hosting service for Git projects, has taken the open source community by storm [26]. At the start of 2015, GitHub hosted over 10M repositories. Compare this with the previous paradigm, centralized version control systems (CVCS), epitomized by SVN [12] and CVS [2]. SourceForge [10], the primary repository hosting service for SVN had about 460K repositories by the start of 2015. Also, our own survey of 820 developers shows that 65% use DVCS and 35% use CVCS.

DVCS brings a whole set of novel capabilities in the area of making and managing software changes. Using DVCS, developers (i) can change and commit code in isolation on local copies of the repositories (without worrying about conflicting changes) while still retaining full project history, (ii) they can cheaply create and merge branches thus allowing them to isolate and group changes by intent, and (iii) they can commit individual changed lines in a file, as opposed to being forced to commit a whole file like in CVCS, thus allowing for a more fine grained selection of the changes that go into a commit.

Are developers truly taking advantage of these DVCS features or are they simply paying the steep learning price without benefiting from them? Despite the large scale adoption of DVCS, we know little about the state of the practice in using this new paradigm. Without such knowledge, developers and managers are left in the dark when deciding whether it is worth investing time and effort to transition to these new tools. Researchers are also in danger of making errors when mining repositories, due to possible confounding effects imposed by DVCS on commit contents. Tool builders can build wrong tools if they are not aware of developers' habits. Without knowing what VCS features the development community uses the most and why, tool builders risk biasing their designs after their own needs. Additionally, without having

a documented record of how developers use software history and perform commits, tool builders risk not providing tool support for important use cases that developers manifest during their work.

In this paper we present a large-scale study that answers in-depth questions about the extent to which DVCS influences the practice of managing changes. We also explore how developers prefer to group changes into commits and how they disseminate a change set with the rest of their team.

To this end, we designed and launched a survey. We recruited 820 participants, 85% of them being developers from industry. 56% have ten or more years of programming experience. 51% work in teams larger than 6 developers. We also interviewed 13 participants on how they manage and communicate software changes, in order to better understand some of the survey results. The participants are software developers recruited from local software companies and graduate students with extensive development experience. On average the participants have 11 years worth of industry experience and work in teams of size 8.

To get further insights into how DVCS affects code changes, we analyzed 409M lines of code changes from 358300 commits. The commits were made by 5890 developers, in 132 repositories containing a total of 73M LOC. Our corpus contains both pure and hybrid repositories. Pure repositories use the same VCS throughout their lifecycle. Hybrid repositories started in the centralized paradigm and were later converted to the distributed paradigm. Hybrid repositories can reveal if changing the version control system influences developers' practices.

For the centralized paradigm we chose SVN as the best representative. For the distributed paradigm we chose Git. Both tools are amongst the most popular representatives for their paradigms.

Using the data from our survey, from mining of repositories, and from developer interviews, we explore the different ways in which centralized and distributed version control systems affect software development and also look into how developers commit and disseminate their changes.

First, we focus on individual programmers and explore how the two version control

paradigms affect the programmer's act of development. One assumption is that the new capabilities of DVCSes affect the way programmers manage their local changes. We explore this assumption by analyzing the differences in programmer committing behaviour when using either CVCSes or DVCSes. In order to better understand gaps in VCS design we also perform a qualitative analysis on how developers split uncommitted changes into commits. Another assumption is that programmers perform certain use cases involving software change that are better supported by DVCSes. We explore this by studying the reasons why programmers prefer one tool over another. The following research questions address this theme:

Theme 1: How does the type of VCS affect developers' behavior?

RQ 1: *Does the type of VCS affect the size of commits?*

RQ 2: *Do developers split their commits into logical units of change? How do they do it?*

RQ 3: *How often and why do developers squash their commits?*

RQ 4: *Why do developers prefer one Version Control System over another?*

RQ 5: *Does the VCS influence the frequency with which developers commit?*

We found that developers' behavior is influenced by the VCS type. When using DVCS, developers make commits 32% smaller and they organize their changes in several commits. Depending on the VCS type, the reasons why developers find the commit process more natural are different. Additionally, we found that developers have a rich set of preferences and behaviours when splitting uncommitted changes.

We also focused on the development process and explore how the use of either CVCSes or DVCSes affect it. More specifically, we are interested in the effects on task management and how the commit size varies across the software development lifecycle. Our assumption is that developers taking advantage of DVCSes can better isolate individual work items. We explore this assumption by analyzing how the presence of issue tracking labels varies with the type of VCS used. We also explore how developers communicate to others that they completed a software change in order to explore new avenues of integration between VCS and other software development tools. The following research questions address this theme:

Theme 2: How does the type of VCS affect the development process?

RQ 6: *Does the type of VCS influence the presence and the number of issue tracking labels (ITL)?*

RQ 7: *Is there a correlation between the number of issue tracking labels in the commit message and the commit size?*

RQ 8: *How does the size of commits vary in time?*

RQ 9: *How do developers communicate to others that they completed a software change?*

We found that developers using DVCS include issue tracking labels more often in commit messages. Also, the commit size decreases as the project matures.

Based on these findings, we propose several actionable implications for four audiences. *Researchers* can better align their research questions with the type of repositories they mine. For example, for questions that rely on a discrete and precise software changes (e.g., bug prediction etc.) they should mine distributed repositories. *Developers* can give more precise meaning to their changes when they use DVCS. *Tool builders* can further build up on the strengths provided by DVCS such as the ability to better group changes and express their intent. *Managers* can make more informed decisions when choosing tools for their projects.

This paper makes the following contributions:

1. **Research Questions.** We designed and answered 9 novel research questions to understand the extent in which DVCS help developers manage software changes.
2. **Survey.** We designed and launched a survey to provide insights into the practice of using DVCS. We recruited 820 participants.
3. **Mining repositories.** We developed tools to collect metrics and analyze centralized and distributed repositories. We applied these tools on 132 repositories.
4. **Interview.** We designed an interview to accompany and further explore the survey topics. We interviewed 13 participants.
5. **Implications.** We present implications of our findings from the perspective of four audiences: *researchers*, *developers*, *tool builders*, and *team managers*.

The tools, summary of survey responses, and corpus are publicly available on the

study's companion website[1].

This thesis is based on our ICSE'14 conference paper [22]. Theme 2 from the conference version has been excluded and other new additions have been made:

- a new source of data, developer interviews, that we use to augment our findings on how developers split changes into commits (RQ 2)
- a new interview based research question on how developers communicate a finished change (RQ 9). We expand our observations, interpretations and implications of the results with the new insights we learned from the interview data.
- a new terminology section on the differences between CVCS and DVCS, and the equivalence between Git commits and SVN commits.
- an expanded tool description

[1] `http://cope.eecs.oregonstate.edu/VCStudy/`

Chapter 2: Terminology

DVCS have a different approach to change management than CVCS. They require developers to learn new commands and concepts. In this section we introduce the concepts that are required to understand the differences that this new paradigm brings. We use terms introduced in a classic book [49] on VCS.

2.1 The centralized and the distributed model

Committing and updating: As Figure 2.1 shows, in CVCS there is one single global repository. This repository holds the commit history of the project and is shared by everybody who works on the project. Developers perform all repository operations concurrently against it.

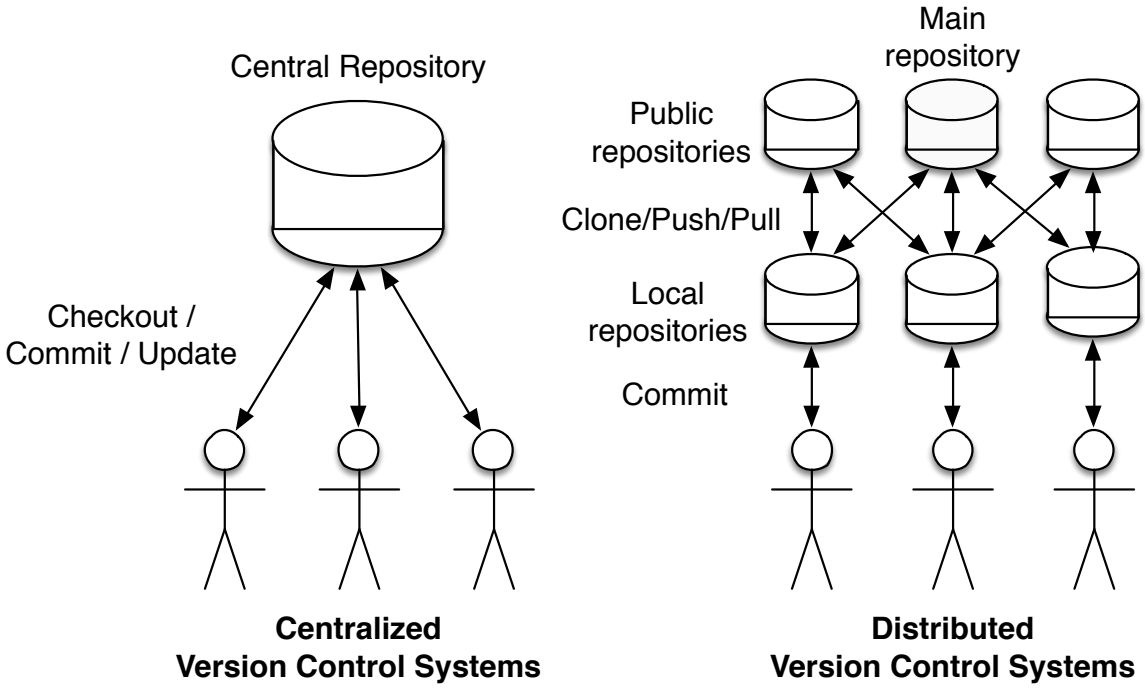
A developer (Alice) that wishes to contribute to a CVCS repository starts by *checking out* the repository. This results in a snapshot (called a *working copy*) of the repository to be downloaded on her machine. The working copy represents a certain revision, usually the latest one. Alice can then make changes to her working copy.

After she has made changes to her working copy, she needs to create a new revision with her changes by performing a commit operation. The commit needs to be applied to the single, shared repository. However, if a colleague (Bob) has already performed other commits before Alice, her commit is rejected, and she is forced to perform an *update*.

Alice performs an *update* to bring her local working copy up to date. If any of the files that Alice changed locally conflict with any of Bob's changes from the remote repository, then she must perform a *merge* before the update can finish.

The centralized paradigm is called an *"edit-merge-commit"* model [49] because it forces users to merge their changes with everybody else's before each commit. The model therefore has a series of disadvantages. First, it forces developers to perform a

Figure 2.1: The centralized and distributed paradigms



time consuming, complicated and error prone merge before each commit [41]. Second, a developer risks losing her pending changes if she fails to perform the merge. This is because she can only commit after successfully completing the merge. Third, in CVCS all repository operations are performed over the network, thus introducing delays in the workflow. These problems are amplified with the increase of the number of developers working on the same repository.

In contrast, if Alice wants to contribute to a DVCS repository, she needs to *clone* the repository. This results in the entire repository, with all its commit history, to be copied over to her machine. In DVCS every user has its own, full, private copy of the repository. All the repository actions are performed on that private copy.

After Alice has made some changes, she commits them. The commit is applied to her private repository and is visible only to her. Therefore she can make as many

and as small commits as she wants without interfering with other colleagues' work and without paying the price of merging after each commit. There is no need for her to worry about conflicts, since she is the only one accessing that repository.

When Alice feels like she is ready to bring in new changes that Bob, her colleague, has done, she performs a *pull* operation on Bob's *remote repository*. This applies any new commits from Bob's repository to her local repository. If any of Bob's changes conflict with her local ones, she would have to manually resolve them.

The main take away idea is that Alice has the power to choose when she is ready to invest effort in merging her team's changes with her own. Moreover, since all her changes are committed (and most DVCS clients force users to commit before pulling), she does not risk loosing pending changes if the merge is not successful.

Performing a pull still does not make Alice's changes public, it just brings external changes to her own copy of the repository. In order to publish her changes and make them public, Alice has to perform a *push* operation on a certain remote repository. The push operation applies any changes on the local repository to the remote one. However, if the remote repository contains new commits that have been made after Alice's last pull, the DVCS client forces her to perform a pull before she can push.

A local repository may have zero or more remote repositories to which a developer can push or pull. However, the development team generally designates one repository that is "canonical", i.e. contains the definitive version of the software.

An advantage of having local history is that it can be modified before pushing it to the central repository. One operation that allows the modification of history is *squashing*. Squashing is the operation where several commits are grouped into a single one. In the process, the initial commits are lost. For example, Alice may perform many small, fine grained intermediary commits when implementing her task in order to help her revert specific changes or keep a mental model of her work in progress. However, these changes are not useful for the rest of the team, so she squashes them in fewer, team relevant commits. We discuss the practice of this type of operation in RQ 3. We also look into the reasons developers use this operation.

Branching model: Besides local repositories, the other major difference of DVCS

over CVCS is that they model history as a *Directed Acyclic Graph (DAG)*: vertices represent commits and edges represent the fact that the source commit is "based on" the sink commit (this is referred to as the sink node being the parent of the source node).

Figure 2.2 represents such a history graph, with three branches. Commit C9 has more than one parent (C5, C6) and represents a merge commit between branches A and Main. Commit C1 has more than one child (C2, C3) and represents a point where history diverges. Branch B contains commits (C7, C8) and has not been merged yet into the Main branch.

Representing history as a DAG allows branching and merging to be treated in a consistent, formal way, and not as mere conventions that have to be followed by developers in CVCS.

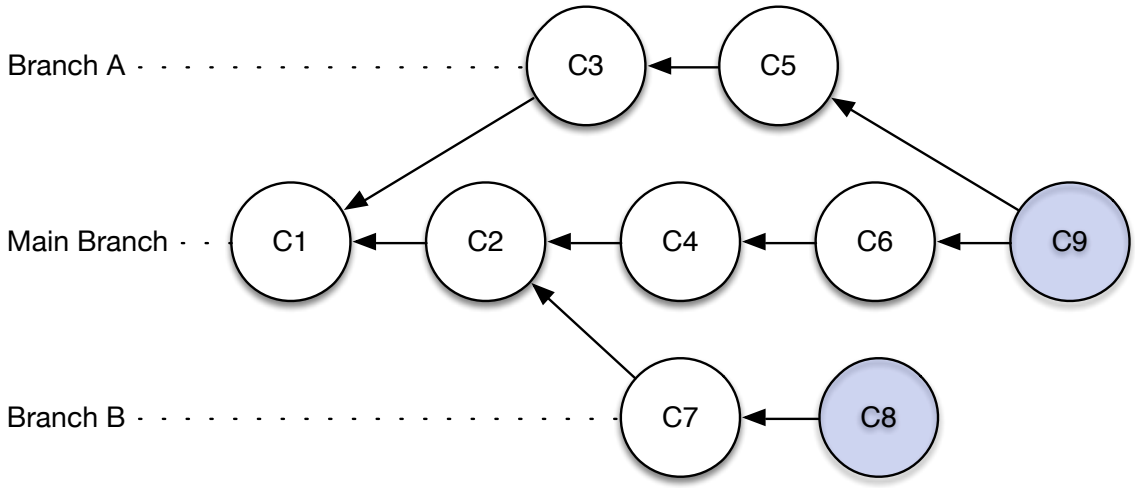


Figure 2.2: An example of a commit history in DVCS.

2.2 Is a Git commit equivalent to an SVN commit?

Given the differences between the two paradigms, the following question arrises: *"Is a Git commit equivalent to an SVN commit?"* We argue that they are equivalent because they both represent the smallest unit of change allowable in the particular

paradigm. In the centralized paradigm, the commit also has the overhead of network communication and possible conflict resolution. When working with a decentralized system, these can be deferred for a later date. Does this difference change the way developers treat software changes?

2.3 Measuring commit size

The *number of lines of code changed (LOCC)* and the *number of files changed (FC)* represent popular metrics for measuring commit size [15, 44]. The FC metric provides a coarse grained, broad brush description of the amount of change introduced by a commit. LOCC provides a more fine grained measurement of the amount of change. For example, two commits that both contain changes to the same file may be arbitrarily different in the amount of change they contain inside that file.

Chapter 3: Experimental Setup

In this section we describe the three sources of data we used to answer our research questions: a survey and interview on developer commit behaviour, and software repositories. The three sources of data are independent of each other.

Table 3.1 shows which data sources contributed to which research questions. The data sources complement one another in providing an overall picture on VCS usage and paradigm shift.

Table 3.1: Mapping of data sources to RQs

Data Source	Research Question
Interview	2, 9
Repository	1, 6, 7, 8
Survey	2, 3, 4, 6

3.1 Survey

We conducted a survey where we asked 20 questions about developer commit practices. 820 respondents answered our survey. The participants are developers recruited by promoting the survey on social media channels specific to the development community, i.e., Twitter and Google+ feeds that are mainly read by developers.

Table 3.2 shows the demographics of the respondents. Most are experienced developers working on industrial projects. The data shows that Git is widely used by developers (52%), followed by SVN (20%).

Classification of open-ended questions: The survey contained both multiple choice and open-ended questions. The contents of the survey can be accessed at the study's companion website [14]. We hand-coded the answers to the open-ended

Table 3.2: Demographics of survey respondents

(a) Programming experience (years)

<2	2 - 5	5 - 10	10 - 15	15 - 20	>20
1.83%	11.10%	30.49%	30.61%	13.90%	12.07%

(b) Project type

Proprietary software	Open source software	Research project	Personal project	Other
85.09%	6.97%	4.64%	3.06%	0.24%

(c) Team size

1	2 - 5	6 - 10	11 - 25	26 - 100	>100
5.87%	42.30%	23.72%	15.65%	8.19%	4.28%

(d) Project age

<6 mo	6 mo - 1 yr	1 yr - 2 yrs	>2 yrs
13.33%	18.58%	21.27%	46.82%

(e) VCS used predominantly

Git	SVN	Hg	Microsoft TFS	CVS	Other
52.68%	20.37%	12.07%	8.54%	1.10%	5.24%

questions using *qualitative thematic coding* [24]. We developed a set of codes that we validated by achieving an inter-rater agreement of over 80% for 20% of the data. Two coders, the second and the third authors, developed the categories which were not known apriori. For measuring the agreement we used the Jaccard coefficient.

Tables B.2, B.4, B.5 show the criteria and definitions that we used for classifying the open ended questions regarding commit splitting, squashing and VCS usability, respectively.

To explore the statistical significance of the difference in responses between par-

ticipants using DVCS or CVCS we used the Chi Square test.

3.2 Developer Interviews

Based on the survey results we wanted to gain further insight on how developers manage their pending software changes and how they communicate to others that a software change has been completed. The former inquiry would help us better define how developers split their changes when they commit and the latter would help us explore new functionalities for VCS. Therefore we interviewed 13 software developers on the two topics.

We recruited participants via a local software development conference [1], Oregon State University's alumni network, graduate students with extensive prior industry experience (10+ years) from Oregon State University's EECS department and via directly contacting companies.

Table 3.3 shows demographic information for the interview participants. On average, the participants are software developers with 11 years of experience and work in team sizes of 8 people. 5 participants use CVCSes at their workplace, 7 use DVCSes and 1 participant uses a mix of both.

We asked the participants to describe how they recorded a recent and an older software change. Table A.1 shows the exact wording of the questions. We described the action of "recording" to the participants as how they split their pending changes into commits and how they use branches. "Software changes" were described as "the set of modifications to the code or tests that would constitute the implementation of a specific requirement, bug fix or other change request". For each of the two changes we also asked the participants how they communicated to others that the change was completed.

The first and second authors of the ICSE'14 conference paper [22] hand-coded the participant's answers using the same technique as the one used with the survey open-ended questions. For questions 1 and 3 we reused the codes from table B.2 to which we added new codes to better capture the interview data (Table B.3). For questions 2 and 4 we developed the codes in Table B.1.

Table 3.3: Demographics of interviewed participants.

Participant	Current job description	Years Professional Experience	Years at current workplace	Usual team size	VCS currently used
P01	PhD Student	12	1	3-6	Git
P02	Software Developer	20	10	8	TFS
P03	Software Developer	21	21	2-4	CVS
P04	Intern	13	3 months	5	Git
P05	Software Developer	4	5 months	5-8	Git
P06	Software Tester	4	3 months	12	Git
P07	Software Developer	3	1	5-6	SVN
P08	Software Developer	9	4	8	SVN, Bazzar, Git
P09	Software Developer	13	13	12-15	TFS
P10	Software Developer	20	3	5	Mercurial
P11	Software Developer	4	7 months	3-10	Git
P12	Software Developer	11	2	5	TFS
P13	Software Developer	10	7.5	10-20	Git

3.3 Repository

To provide further insights into how DVCS affects developer's practices, we collected and analyzed 132 software repositories.

3.3.1 Repository Corpus

To answer our research questions we needed to collect software repositories that are representative of the centralized and distributed paradigms. We also collected hybrid repositories that started in a centralized paradigm and switched to the distributed one. Our assumption is that differences in metrics taken from these 3 kinds of repositories provide valuable insights on how they influence source code management.

Using the survey results, we selected SVN and Git as being representative for the centralized and distributed categories, respectively. Thus we collected three kinds of repositories:

1. Pure SVN repositories: repositories with commits originating in SVN.
2. Pure Git repositories: repositories with commits originating in Git.
3. Hybrid repositories: repositories that were originally SVN repositories but were

converted by developers into Git repositories via repository conversion tools such as svn2git [13]. We partitioned the commits of hybrid repositories in two chronological stages: an initial stage that contains SVN commits (that developers performed while the project was using SVN) and a subsequent stage that contains Git commits (that developers performed after the project switched to Git). We refer to the two stages of hybrid repositories as $\text{Hybrid}_{SVNStage}$ and $\text{Hybrid}_{GitStage}$.

We collected SVN repositories from SourceForge and Git repositories from GitHub. These repositories span several programming languages: Java, C, C++, JavaScript, and Python.

For GitHub we selected the top ranked repositories, i.e., repositories that have been marked as favorites by developers and/or have been forked the most. For SourceForge we used its own internal ranking metric to select the top ranked repositories. We queried the SourceForge projects through the Notre Dame Sourceforge Research Archive [11], which serves as a mirror designed specifically for researchers. By choosing the top repositories we ensure that we collect mature projects with a rich history.

To find hybrid repositories, we searched for internet posts about migrating repositories from SVN to Git, assuming that the post authors would have done this conversion themselves. For each post that we found we searched the author's public code repositories for hybrid repositories. In addition, while collecting pure Git repositories, some of them proved to have actually started in SVN. We classified these repositories as hybrid repositories and not pure Git repositories.

For each hybrid repository we manually identified a switch-over threshold as the timestamp when the repository switched to Git. We accomplished this by searching for specific template strings that conversion tools such as svn2git [13] insert in the original SVN commit messages or by searching for Git specific events, such as the first commit to reference the ".gitignore" file. Then we tagged all commits performed at a date smaller than the threshold as SVN commits and all commits performed at a date larger than the threshold as Git commits. The study's companion website

Table 3.4: Repository corpus.

Repo. Type	Repositories	Commits	Authors	Total LOC changed
SVN	52	95571	451	270M
Hybrid	29	151004	2249	89M
Git	51	111725	3190	50M
Total	132	358300	5890	409M

contains information [14] on actual switch over dates.

We took extra care to ensure the integrity of repositories, i.e., Git repositories did not originate in SVN, by searching for keywords in commit messages. Also, the pure Git repositories are independent (i.e., there are no repositories that are forks of each other).

Table 3.4 shows the corpus of repositories. For each repository kind, we tabulate the number of individual repositories, commits, and authors that contributed. The last column shows the total number of lines of code that have been changed by all commits. We defined an author as a committer who performed more than 4 commits.

47% of the repositories represent software libraries (e.g., GUI components, utility functionality, program analysis, IO, etc). 34% represent domain specific applications (e.g., task management, business software, games, specialized computation, code editors etc). 19% represent software frameworks (e.g., embedded applications, generic content managers, program analysis, persistence, etc). The study's companion website contains a list of individual repositories [14].

We aimed for an equal number of SVN and Git repositories to ensure that we compare the two paradigms in a fair way.

3.3.2 Repository Analysis

We used Git as the canonical representation for all repositories. This is possible since the Git object storage model is a superset of the centralized model. For example, the linear history of CVCS can be easily represented in Git's directed acyclic graph branching model. Thus, we converted all SVN repositories to Git, using the SVN2GIT tool [13].

To explore the statistical significance of various sample differences, we applied the Wilcoxon rank-sum test. We chose this test since Wilcoxon rank sum is robust against non-normal distributions of data, and we were unsure about the normality of the population from which we sampled.

We used the Pearson correlation coefficient in order to establish linear dependence between two sets of randomly distributed values.

3.3.2.1 Tool description

We have built an analysis platform to gather several commit metrics. Figure 3.1 shows the complete tool stack.

Our platform is built over GITECTIVE and implements a configurable analysis pipeline[1] that visits each commit in a given repository. The pipeline first filters out unwanted commits and then collects measurements of interest from the remaining ones. It can be configured[2] with any custom made commit filters and data collectors. Additionally, the pipeline can be configured with a custom data aggregator[3] (e.g., aggregate data in CSV, JSON, database, etc). Our tool also extends Gitective with the ability to reject individual diff regions (e.g., reject code formatting diffs, or comment editing diffs[4]).

GITECTIVE [4], is a repository analysis platform built on top of JGIT. By sublclassing *RevFilter* from JGIT it enhances the traversal resolution, allowing clients

[1] PipelineCommitFilter
[2] PipelineCommitFilter.addFilter(CommitFilter), addDataCollector(CommitFilter)
[3] PipelineCommitFilter.setDataAgregator(AnalysisFilter)
[4] FormatEditFilter, CommentEditFilter

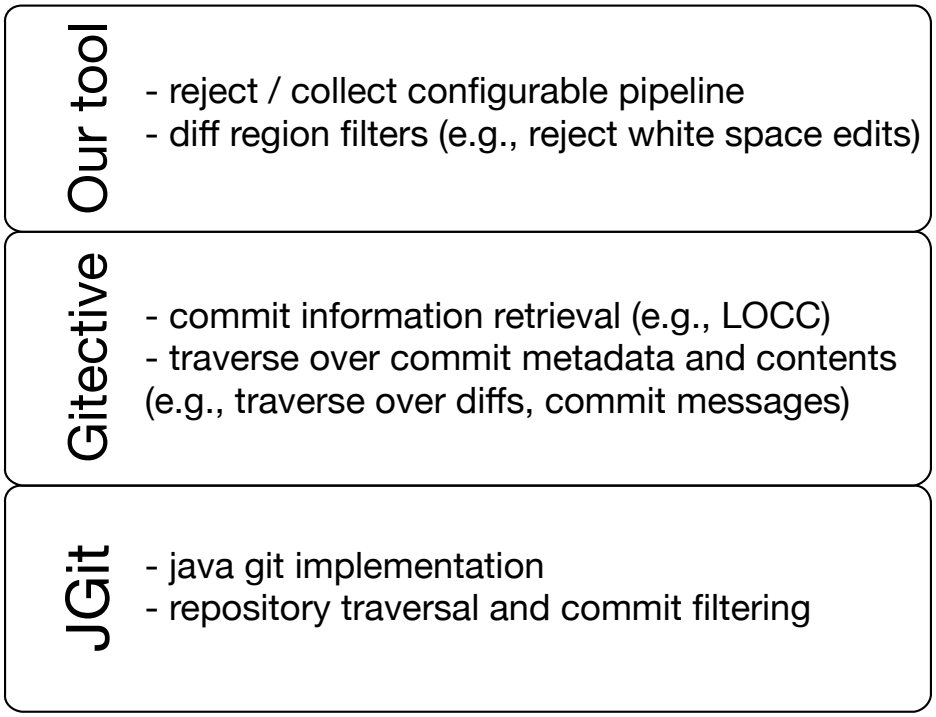


Figure 3.1: Tool stack

to traverse over commit metadata (e.g., commit author, date, message, etc) and even individual diff regions. In addition to traversal, it also computes several metrics such as LOCC.

JGit [8] is a java implementation of Git. It provides an API for all the Git commands. Additionally, its API provides a means of traversing the commit tree of a repository, one commit at a time. By subclassing *RevFilter* clients can specify their own rules for excluding commits from the traversal.

3.3.2.2 Filtering changes and metric collection

Filtering changes: Our initial manual investigation of commits suggests that many commits do not represent actual programming changes carried out by developers (e.g., changing features, bug fixing, refactoring, etc.), but are the result of applying tools

such as code formatters. Such commits are extremely large, i.e., they affect thousands of LOC. Since these commits would bias our analysis, we decided to filter them out.

Moreover, we are interested in investigating only commits that change existing code and not commits that add or remove code. We do this primarily because we are interested in how developers manage pending changes to an existing code base and because additions and insertions can be arbitrarily large.

Our analysis filtered out any commit that:

- consists only of either added, deleted or renamed files. Most of the times these commits represent large scale project file structure modifications. Also, commits that only add files do not interact with any part of the program and were therefore eliminated.
- is a merge commit. These commits usually represent decisions on conflict resolution and contain changes from several lines of development.
- updates only copyright notes, code documentation (e.g., JavaDoc comments) or reorganize code dependences (e.g., `import` statements).
- is artificially created by repository migration tools.

Inside each commit, we ignore all changes that modify only comments and white spaces. To further protect our measurements from the noise of large commits, we further retain the commits bellow the 90th percentile in regards to LOCC after applying the specialized filters from above.

For each commit we collect the following metrics:

Commit id, for identifying commits.

Commit date, for sorting commits chronologically.

The author of the commit, for grouping by authors.

Number of LOC changed by the commit (LOCC), for determining the size of commits. For each commit we compute LOC added, deleted, or modified as reported by the standard DIFF tool.

Number of files impacted by the commit (FC), for determining the commit size. While LOCC tells us how much software editing has been performed in a commit, the number of impacted files tells us how spread the change is within the system.

Number of issues referenced in the commit message, to determine the cohesiveness of changes. The issues refer to programming tasks, such as features or bugs, managed with external systems such as BugZilla, Jira [34], etc. In order to detect them, we used an approach similar to the one described by Bird at al. [19], which employs searching for specific text patterns in the commit message: our tool includes an algorithm that searches commit messages for the presence of words and word combinations such as "bug/s", "issue/s", "task/s", "ticket/s", "feature/s", "fixed by", "closes", "completed" and others, followed by a number or several numbers separated with "and" or comma. The algorithm is also aware of specific issue referencing formats. (e.g., AAA-123 for JIRA or #123 for GitHub). Patterns which would cause false-positives are excluded (e.g., patterns that represent software versions, dates, etc). Using the described heuristic, the tool computes the total number of issue references found in a commit message.

Chapter 4: Results

This section presents answers and insights on the study's 9 research questions, based on our three sources of data: surveys, developer interviews, and repository data. Each question presents our analysis results and our interpretation of the results. The questions are grouped in two major themes. At the end of each theme we present implications and advice for researchers, tool builders, software developers, and team managers.

4.1 How does the type of VCS affect developers' behavior?

RQ 1: Does the type of VCS affect the size of commits?

Table 4.1 shows the commit size, in terms of LOCC and FC, made by individual authors. The data is grouped by VCS type.

Table 4.1: Commit size across different VCS in terms of source lines changed (LOCC), and files changed (FC)

	Mean		Median		StdDev	
	LOCC	FC	LOCC	FC	LOCC	FC
Git	27.20	3.08	13.46	1.96	32.72	2.7
Svn	40.06	5.65	18.44	3.19	49.62	6.72
$\text{Hybrid}_{GitStage}$	23.02	2.40	11.52	1.70	27.57	1.74
$\text{Hybrid}_{SVNStage}$	25.72	2.82	12.61	1.96	31.24	2.15

In terms of LOCC, the commits from Git repositories tend to be smaller than those made in SVN repositories ($p < 0.01$). The mean and median LOCC in Git repositories is 27.20 and 13.46, respectively, while for SVN repositories these values are 40.06 and 18.44 respectively. The standard deviation of LOCC also differs, with

32.72 LOCC for Git and 49.62 LOCC for SVN. The populations under test are the means, medians and standard deviations of commit LOCC by authors and grouped by VCS type.

In terms of FC, the same trend of a smaller commit size can be seen as with the LOCC metric, although the difference is not significant ($p > 0.05$). Commits from Git repositories tend to affect fewer files than commits from SVN repositories.

On the other hand, for hybrid repositories our data was insufficient to differentiate between the commit size of the Git and SVN stages, in terms of LOCC ($p > 0.05$).

Observation 1: DVCS repositories have a smaller commit size than CVCS repositories.

Observation 2: Results are not significant enough to state whether hybrid repositories have a smaller commit size after switching to DVCS.

Interpretation: One possible explanation for Git commits being smaller than SVN commits is the fact that Git enables its users to select finer grained changes to commit. In Git the atomic unit of change that can be committed is the line while in SVN it is the file.

Another possible cause that enables small commits in Git is that each developer commits to his own local repository without the need to synchronize with everybody else. This means that there is no risk of conflicts upon every commit.

One participant stated that *"Git promotes the idea that your commit space is not inflicting pain on anyone else, so frequent commit and experimentation is encouraged. By design it promotes small, frequent commits that serve a specific purpose rather than the '5pm commit."'*. Resolving conflicts becomes a task that is consciously entered into when deciding to synchronize changes with other team members. It is not something that must happen with every commit.

We did not find a significant difference of commit size in hybrid repositories. This could mean that the culture of a project can take a long time to change when a new tool is introduced. Thus, in long lasting projects, it seems that old habits die hard.

RQ 2: Do developers split their commits into logical units of change? How do they do it?

The changes that a developer makes might belong to one or more logical units of change. Do developers split these changes and commit separately? Or do they just group everything and generate one large commit? We used the survey to obtain an initial, broad-brush picture and then augmented our findings with in-depth interviews.

The answers in the survey give us the picture depicted in Table 4.2. While both DVCS and CVCS participants tend to split their uncommitted changes into multiple commits, the percentage is higher for DVCS participants ($p < 0.05$).

Table 4.2: Developers splitting their commits (%)

Practice	DVCS	CVCS	Overall
Split their changes	81.25	67.89	75.99
Group their changes	12.50	26.61	18.05
Other	6.25	5.50	5.96

Observation 3: 76% of the developers split their commits. The percentage is higher for distributed version control systems (81.25%), compared to centralized ones (67.89%).

A question of great interest is the criteria on how they split their changes. We chose four categories to capture the respondents' answers:

Implementation details refer to *how* was a change carried out (e.g., change field type, add new branch to a switch statement, etc). *Intent of change* splits changes by expressing the *what* part of the change carried out (e.g., add a feature, fix a bug). *Policy* splits changes based on a criteria that is externally imposed (management practices, development process, etc). *Other* represent reasons that do not fit in the above criteria. Table B.2 shows the actual codes we extracted from the open ended survey question (Q10) along with response examples for each code. Splitting by implementation reasons corresponds to the fine-grained scope and splitting by the intent of change corresponds to the coarse-grained scope.

Table 4.3: Reasons for splitting commits (%)

Technique	DVCS	CVCS	Overall
Implementation details	37.01	21.85	32.03
Intent of change	45.13	62.25	50.76
Policy	6.17	5.30	5.88
Other	11.69	10.60	11.33

Table 4.3 shows the distribution of splitting reasons for each VCS paradigm. We observe that in the case of DVCS, developers split their changes based on implementation details more frequently than they do in CVCS ($p < 0.05$). This will inevitably result in more commits.

Observation 4: Overall, developers choose to split their commits using the intent of change.

Observation 5: More DVCS users split changes based on implementation details than CVCS users.

As we have seen in observations 3 and 5, DVCS users split their changes in several commits more often and they do it with a finer-grained scope in mind. One participant reported: *"Each commit is one cohesive change that might fix a bug, add new functionality, alter existing functionality ([...] like "sphere class can now calculate its own volume" - user level features usually take many commits)"*. This corroborates with the findings about the influence of Version Control Systems on commit size (RQ 1). Being able to more easily split the commits and the commit process being simpler as well, will result in smaller commit size.

We now use the interview data to further explore why developers choose to split their changes to express either implementation details or the intent of change. The additional interview codes that refer to splitting commits are shown in table B.3.

A reoccurring reason for splitting changes in order to express implementation details (fine grained commits) is keeping changes untangled. One benefit of keeping changes untangled is being able to better roll back specific changes. One participant

states that: *"I prefer smaller topic commits because that captures the history of development a little bit better. Not always, but, it also makes it easier to prune out changes that were not necessarily beneficial"*.

Another benefit of untangled changes is having an easier to read and understand development history. A participant provided a generic example where having a big commit with tangled changes harms software history understandability. She described a recurring need to go back in history and understand the changes required to fix an old bug. Usually the bugfix is contained by one or more commits. However, if the commit containing the bugfix includes other changes such as refactorings or formatting it becomes a time intensive activity to understand what the change required to fix the bug was. She concludes that *"For us it is very much of matter of being able to focus on the reason for each line being changed when we go back through history"*.

However, choosing to commit very fine grained implementation details may make software history convoluted, obscuring the high level changes: *"... But then you get into this dependency graph of changes that is not apparent just by looking through the serial history of commits. Topical commits are not always the best I guess"*. A way participants got around this problem is to have branches represent the high level intent of the change and keep commits granular with implementation details: *"We created a separate branch in git and then we were using disconnected local branches for people to make many small changes that added up to one big functional change and then we would do [GitHub] pull requests to pull them back in"*.

Observation 6: Some developers like to keep untangled changes by making small, granular commits. This helps them in searching, manipulating and understanding past changes.

The two most common interview arguments for making large feature commits refer to avoiding to commit changes without corresponding tests and changes that break the build. As one participant states, *"What I would not want to do is commit something that would cause functionality to break such that if other developers were to get that particular version before I get the next commit done, the system would not work for them. So if that is the case, then I will hold on to it and not commit until*

I have functioning version that people can build on top of and then I would commit". A participant whose team uses a CVCS stated that large feature commits are easier to review because they contain all the related changes in reference to a programming task: *"I think that it helps reviewing because you can open the change set and you can see all the corresponding things that have changed as part of that change set. It is easier for the reviewers to coordinate that set of changes and pull them together".*

We learned that some participants use an intermediary solution between fine grained logical commits and coarse grained feature commits: they try to divide a high level programming task in smaller high level units of functionality that can correctly function in isolation. One participant states that: " *We try to compartmentalize the feature updates as small as possible. You might have an overall user story like [project sensitive details]. This was actual feature they gave us. That particular feature, you might decide to "I will just go ahead and do the menu part ... We usually compartmentalize to the point where each piece is fully functional to the point they can make it. So the menu system might be developed to where it does not actually do anything but it functionally works ... We don't break the code. We do not want the checkin ever in a state where I cannot download it and run it".*

Observation 7: Some developers like to group changes by intent, thus making large, self contained commits.

Sometimes participants expressed conflicting needs in regards to commit size. One interview participant described an instance where he had made 7-8 small commits but was asked by his task reviewers to merge them into 2-3 more cohesive commits. Another interview participant described that in his company there are QA engineers who watch the main branch for commits in order to review each change. Their requirements are that each commit should compile and have tests. However, the participant needed to collaborate and share his intermediate work with another developer: *"I am usually pretty uneasy about what I want to put in a commit because there are some people that feel that everything that is committed should compile and have running tests all the time. But that kinda limits how often you commit things because then you won't end up committing a day's worth of work in some cases. For a couple of*

days I had several broken tests that I didn't have time to go and fix and at some point I had something that wasn't compiling that I needed to share with another developer". The participant's solution was to make a branch on which he could perform small code sharing commits: *"The reason we went off onto this project specific branch for this refactoring was because we knew that it was going to be more than a day's worth of work where we did not want to commit it into the repository that QA was going to be working from, and we weren't ready to integrate our work. So we worked on this project specific branch until we were ready to integrate back".*

Observation 8: There exist conflicting needs inside software teams in regards to commit size and contents.

Interpretation

One explanation for observation 3 is that in DVCS, the commit process is easier and cheaper than in centralized ones. There is no risk of conflict with each new local commit. Moreover, the smallest atomic unit of change in DVCS is the line, not the file (as it is in CVCS). All these make committing easier, so developers are willing to take the time to split and commit each logical change separately. In a recent study conducted in parallel with ours, Muslu et al. [43] have also discovered that the ability to commit locally and independently allows developers to work incrementally. This interpretation is also suggested by observation 5: an easier and cheaper commit process allows DVCS users to express change at a more granular level, thus enabling them to capture implementation details.

Based on developers' responses in choosing between small fine grained commits at the implementation level and large coarse grained commits at the intent of change level we conjecture that developers would prefer having both capabilities at the same time: a means to record and display both fine grained implementation details and coarse grained feature level details. Developers use Git and Github complementary: they express implementation level details via Git's support for small fine grained commits and high level change intent details (e.g., features, bugs, etc) via Github's pull request feature. On the other hand, developers that use VCS tools which do not support both levels of detail have to choose between the two.

RQ 3: How often and why do developers squash their commits?

Results from the survey show that only 30% of the developers squash their commits. The results for the distributed and centralized repositories are shown in Table 4.4.

Table 4.4: Developers squashing their commits (%)

Response	DVCS	CVCS	Overall
Yes	36.59	18.12	30.21
No	54.79	44.57	51.31
Not applicable	8.62	37.32	18.48

Table 4.5: Reasons why developers squash their commits (%)

Reason	DVCS	CVCS	Overall
Group similar changes	25.63	45.16	28.80
Intermediate steps are irrelevant	20	0	16.75
Remove mistakes	15	0	12.57
Keep history clean	26.88	6.45	23.56
Policy requirement	5.63	9.68	6.28
Other	6.88	38.71	12.04

Table 4.4 shows that squashing happens twice more often in distributed repositories than in centralized ones[1]. This probably has to do with the fact that it is easier to manipulate commits in DVCS. Developers who practice squashing mention two main reasons (Table 4.5): (i) to group several changes together and, (ii) in cleaning history they do not care about the path they took to a solution as long as it's finished and it works.

To provide more insights into how we summarized the data in Table 4.4, we present Table B.4 that shows the extracted codes from the open ended survey question along

[1]See Internal Threats to Validity (Section 5)

with descriptions and response examples for each code.

Observation 9: Squashing does not occur often in practice. If it does occur, it's a practice mainly associated with DVCS

RQ 4: Why do developers prefer one Version Control System over another?

According to the survey, we have found two main reasons why developers find a commit process more natural. The first is the presence of a *killer feature*. It usually helps developers achieve higher productivity by allowing a workflow that is more comfortable for them. The second is habit. Developers get used to a certain tool. Therefore, they will find the tool natural to use from the habits they have acquired while using it on a daily basis. Table 4.6 shows the distribution of reasons for tool preference across the two VCS paradigms.

To provide more insight into how we derived developers' preferences we present Table B.5 that shows the extracted codes from the open ended survey question along with descriptions and response examples for each code.

Table 4.6: Reasons for considering a VCS more "natural" to commit (%)

Reason	DVCS	CVCS	Overall
Killer feature	46.02	10.89	30.41
Old habit	22.88	41.58	30.41
Easy to use	19.79	41.58	27.14
Personal preference	2.06	0.99	2.04
Other	9.25	4.95	10

In 46% of the cases developers prefer DVCS because of a *killer feature*. By looking at individual replies we have found that one of the features mentioned is the possibility to commit to the local copy of the repository. Also, we can see that the main reason for preferring CVCS is the ease of use. While the distributed model has its advantages, that comes at the cost of a more complex model. This could explain why

so many developers (almost 42%) think that the centralized model is easier to use. The difference between DVCS and CVCS responses is significant ($p < 0.05$).

Also, many prefer CVCS simply because of habit. Having used a system for a very long time, one gets accustomed with its command interface and paradigm. It is interesting to note that CVCS are used not for their capabilities in managing change, but due to old habits and a faster learning curve.

Next we describe the killer features that our participants referenced for DVCS. By far, most respondents preferred the ability to work on local copies, with 45 of the respondents mentioning it. One participant stated that *"You get to commit to a local repository and make your changes public only when they are ready."*

The second most preferred feature was the ability to select files to commit. 30 participants mentioned this in their replies. One participant stated that they think that DVCS feel more natural *"Because you get to choose which files [go in] the commit."*

Third, our participants liked the ability to modify history, before making it public. 20 participants stated that they found DVCS more natural because of this. One participant stated that *"[. . .] you can still change your commits as long as you have not 'published' them by pushing them upstream."*

16 participants mentioned that they considered branching and merging as the reason they felt that DVCS are more natural. One partipant stated that *"Merging is less painful."* Another stated that branching was cheaper: *"The merging, branching, [. . .] functionalities are extremely fast and convenient."* Other participants liked that the branching model in DVCS maps better to their workflow, as exemplified by the following two quotes: *"The branching model best matches how I spend my time."* and; *"Branching [. . .] mirrors how [my] thoughts work[,] where I can have mainline work but [I can] go on tangents without losing my place."* Also, one participant stated that *"Branches are built into the tool instead of them being conventions, [. . .]."*

Observation 10: The commit process of DVCS is perceived by developers to be more natural because of the presence of killer features.

Observation 11: The commit process of CVCS is perceived to be more natural because of familiarity and a faster learning curve, not their feature set.

Our findings are reinforced by Muslu et al. [43]. Their study shows that developers prefer the ability to work offline. Also, they have found the learning curve to be a barrier in adopting DVCS.

RQ 5: Does the VCS influence the frequency with which developers commit?

Table 4.7 shows results we obtained from the survey. Developers commit several times a day regardless of the version control they use. The data for each VCS type shows a slightly different picture. Developers using DVCS commit once an hour more often (19.66%) than developers using CVCS (4.10%). Also, when using CVCS developers are more likely to commit once a day (14.75%) than when using DVCS (7.19%). The difference in commit frequency is significant ($p < 0.05$).

Observation 12: Most developers have similar commit frequency habits independent of what VCS they use. However, the percentage of developers that commit at the hour and minute levels is higher for DVCS users than CVCS users.

Table 4.7: How often do developers commit? (%)

	DVCS	CVCS	Overall
Once a minute	3.38	0.82	2.51
Once an hour	19.66	4.10	14.37
Several times a day	65.96	66.80	66.25
Once a day	7.19	14.75	9.76
Several times a week	1.90	9.43	4.46
Once a week	1.48	3.32	2.09
Once a month	0.42	0.82	0.56

Interpretation: The fact that developers commit once an hour more often when using DVCS than when using CVCS suggests that they find it easier to commit. Results from the previous research questions also lead to this conclusion. One interesting results is that 14.75% of developers using CVCS commit once a day. This suggest a

pattern of committing once the work day is over.

4.1.1 Implications

For developers: Smaller commits make code reviews easier. Having a tool that enables small, fine grained commits allows users to separate and document each change individually. One participant mentioned that they split their commits because *"[changes] should be logically separated, to easily allow [the] commit message to drive [the] review"*. Consider reviewing a new feature that has been added. Instead of going through thousands of changes, the reviewer can go through one change at a time, each explained by the commit message.

Also, smaller commits enables easier bisecting. This enables techniques such as Delta Debugging [55] to be employed to find the root cause of bugs.

The concurrent programming model that is enabled by DVCS also brings new overhead in managing and synchronizing with remote repositories. This makes Git harder to learn and master. Thus, Git has a more steep learning curve.

Using a DVCS can offer developers more power when it comes to choosing what to commit. DVCS tools like Git allow the splitting of commits at line level, which helps when changes with multiple intents are interleaved in a single file. This kind of separation is not possible when using SVN. A participant mentioned that he preferred Git because *"it gives useful tools for splitting or merging commits"*.

By splitting changes into multiple and smaller commits developers can cherry-pick changes. Cherry-picking refers to the operation of selecting one commit from a branch and applying it to another one. This way, developers can migrate changes from one branch to the other without the need to merge all changes. This has maximum benefits when commits carry only one intent, as noted by one respondent who splits his commits because of *"the ability to easily cherry pick or revert [commits]"*.

Developers can remove mistakes and clean a project's history by squashing their commits. Several respondents mentioned that they squash to *"To correct a previous commit"* or *"To make it easier for people reading the log to understand what's been*

changed". However, we see in observation 9 that it is not widely used. This is because, sometimes, squashing leads to a loss of historical data. This information might be useful in the future when debugging or trying to understand the origin of some changes.

From observation 10 we learn that developers like DVCS because of some of their *killer features*. One that was mentioned often was the ability to commit locally: *"You get to commit to a local repository and make your changes public only when they are ready"*. Learning how to use these features takes time and effort. Using the same tools allows developers to keep their level of productivity in the short run. However, the initial effort and loss of productivity caused by learning a new version control system or paradigm may pay off in the long run. One participant reported that he *"tried Git but its too similar yet just different enough to confuse the hell out of me and slow us all down"*. Another *"[...] was not happy about this [using Git] to start off with, and it took me about two years to learn and love Git"*. The advantages of switching would be overall increased productivity, compared to using a CVCS, and better history and management of software changes.

For researchers: Researchers mining software repositories and studying discrete changes should focus on DVCS because they allow smaller atomic units of change. For refactoring researchers, the smaller Git commits could better define individual changes. For researchers who tie different software artifacts to code, such as bugs, Git commits are more precise therefore they may have fewer false mappings.

Researchers must be careful when collecting software repository related metrics. Old repositories that migrated through several VCS tools present a different behavior than pure repositories. It may be the case that the size and structure of commits vary with the VCS tool in a hybrid repository. There might be other phenomena that influence a repository's structure. By not paying attention to different phenomena that affect repositories researchers risk biasing or confounding their results.

There is a lot of noise when studying different types of software changes introduced by commits. As seen in section 3.3.2, there are many types of commits and individual changes that do not constitute acts of development. Researchers should clearly define

what types of changes they are studying and then take the appropriate actions to filter undesired commits. By not paying attention to different types of commits, researchers risk biasing or confounding their results.

DVCS allow users to change history before they make it public or available to others. One participant stated he squashed commits because he *"committed more often locally while working. That need not be seen in the final push, because it usually only adds noise"*. This is a threat when mining repositories. The repository that is publicly available might not be the one that developers had when they committed their changes. Squashing is just one of the ways in which developers can change history. Research on such repositories should take this threat into account.

For tool builders: Although Git enables finer grained changes, it is still the developers' task to disentangle these changes. This is a manual, tedious, and time consuming process. VCS tools could keep track of different change intents and then offer to commit them separately. Herzig et al. [31] and Dias et al. [25] show techniques by which this can be achieved. They devise heuristic untangling algorithms that split tangled changes according to different source code criteria (e.g., the distance between two changed AST nodes, the temporal distance between two edits, etc).

We envision a new generation of tools that can use the average size of a commit as a quality metric. When a developer has uncommitted code larger than a threshold, the tool could suggest that it's time to split changes and commit.

Continuing on the idea of metrics, the field of software design flaws can be applied to repositories as well. Researchers have identified many software design flaws [37]. Marinescu [38] presents detection strategies for these flaws, allowing tools to identify, report and offer suggestions for improvement. By following this approach researchers can devise design flaws for repositories and then metric based detection strategies for these flaws would allow tools to measure the health of a repository.

Developers like having small commits containing untangled changes which ease the understandability of software history. On the other hand, they also like big feature-level commits to express high level software evolution history. With today's VCS tools developers either have to sacrifice one capability over the other (e.g., SVN), or

have to use a combination of VCS tools and graphical clients (e.g., Git and Github) in order to express both. We suggest that VCS builders should design and implement change models capable of representing both levels of granularity. Such a model might be the ability to group small fine grained commits that express implementation level details into virtual composite commits that express higher level change intents.

Squashing is a process by which history can be altered or completely lost. To prevent history loss, VCS tools could support features such as hierarchical commits: the ability to create a virtual commit that holds other real commits. Instead of loosing history through squashing, developers could group commits into larger, composite commits.

Observation 2 suggests that changing the paradigm may not change developer behaviour. We need educational materials to teach people how to adopt the new system's capabilities.

Respondents identified features as an important factor for using DVCS. Some mentioned that certain features were an integral part of their workflow. Paying attention to these workflows and creating the tools to support them will pay off in the future. The payoff will increase productivity on the developers side, and bring a larger user base on the tool builder's side, since developers will prefer a tool that best fits their work style.

For team management: As Observation 2 states, hybrid repositories may not show the same trends as non hybrid ones. This shows that adopting new tools and new technology is only part of the change and by no means enough or complete. Tools that bring a new vision to how software is developed should be followed by a shift in policy and project culture as well. One cannot hope to improve the development process by only improving the tools.

4.2 How does the type of VCS affect the development process?

RQ 6: Does the type of VCS influence the presence and the number of issue tracking labels (ITL)?

The survey shows that the majority of developers (69%) commit only one issue at a time (Table 4.8). This figure is slightly smaller for CVCS than it is for DVCS. However, CVCS developers perform commits with more than one issue more often (17%) than developer using DVCS (9%).

Table 4.8: Survey: How do developers commit if they work on more than one issue (%)

	1 issue	>1 issue	Not applicable
Distributed	68.71	8.59	22.7
Centralized	66.67	17.03	16.03
Overall	69.25	11.13	19.13

The repository analysis shows that 31% of all commits contains an ITL ($p < 0.05$). This mirrors similar findings by Bird et al. [19]. The number is higher for Git repositories at 43.42%. For hybrid repositories the number is at 33.12% compared to SVN at 13.13%.

Observation 13: A small number of commits are labelled with ITL. Nevertheless, issue tracking labels appear more frequently in DVCS commit messages than in CVCS commit messages.

Interpretation: The fact that we see a smaller number of developers committing changes belonging to two or more issues per commit for CVCS might be an indication of a higher difficulty in selecting the changes to be committed. The difference in the granularity of change selection between the tools could explain these results.

The overall low number of commits with ITL can be attributed to a relaxed commit policy. As the repositories are gathered from open source projects, it may be difficult to enforce a strict commit policy. To the best of our knowledge, none of

the analyzed repositories enforced a practice of mandatory ITL inclusion in commit messages.

RQ 7: Is there a correlation between the number of issue tracking labels in the commit message and the commit size?

We can see that commits to SVN and Hybrid repositories tend to be larger when more issues appear in the commit message. The correlation is strong and positive, at 0.68 for SVN and 0.81 for Hybrid repositories. For Git repositories this trend does not hold. There is a slight tendency for commit size to decrease when the number of issue tracking labels increases. There is a weak negative correlation, at -0.38). Table 4.9 shows the detailed results.

Table 4.9: Average LOCC for issue references by VCS type

VCS type	number of issue references					corr.
	1	2	3	4	5	
SVN	33.04	35.69	54.56	31	80	0.68
Git	25.27	36.46	38.05	23	23	-0.38
Hybrid	27.67	31.66	37.74	83.2	62.14	0.81
All	28.59	34.72	39.91	57.78	60.08	0.97

Observation 14: In SVN and Hybrid repositories commit size is positively correlated with the number of referenced issues.

Observation 15: In Git repositories there is a weak negative correlation between the commit size and the number of referenced issues.

Interpretation: The strong correlations for SVN and hybrid repositories reinforce the idea that in these repositories developers tend to group different change intents (issues) together: the more issues a commit addresses, the larger the commit is.

In Git, the trend seems to be opposite. This suggests that Git commits do not get larger in size when they reference several ITL. Rather, Git commits could contain a change common to all referenced issues.

Observations 3 and 5 show that developers using DVCS split their commits more often and that their commits contain finer scoped changes. This hints to the idea that they might carve out the common piece of code that contributes to solving both issues.

The fact that hybrid repositories show the same trends as SVN repositories, together with observation 2, makes us hypothesize that in hybrid repositories old commit splitting habits die hard despite the fact that developers tend to perform smaller commits after transitioning to DVCS: if developers used to group different changes together in SVN, then they might do the same even after they migrate to Git.

RQ 8: How does the size of commits vary in time?

In order to investigate how the commit size varies in time, we averaged the commit size for monthly intervals. We then calculated correlation coefficients for the monthly values of average commit size.

Table 4.10 shows that the commit size tends to become smaller as projects get older. The average age of a typical repository (time between the first and the last commit) is 55 months. For SVN repositories it is 54 months, for Git repositories it is 30 months and for hybrid ones it is 94 months.

Table 4.10: Correlation between commit size and commit time

VCS	average corr.	# of positive corr.	# of negative corr.	% of negative corr.
SVN	-0.06	21	31	60%
GIT	-0.17	13	25	66%
Hybrid	-0.11	12	16	57%
ALL	-0.11	46	72	61%

The average commit size usually decreases by approximately 15-20% during the

lifetime of a repository. Overall correlation between commit size and time of commit for all types of repository is usually negligible (-0.11) and in most cases appears to be negative.

Commit size tends to decrease more in Git then it does in SVN. 66% of the analysed Git repositories show decreasing trends in average commit size over time. For SVN only 60% showed this trend, while for hybrid repositories this number constitutes only 57%.

Observation 16: Commit size tends to become smaller as projects get older.

Interpretation: This decreasing trend can be explained by different types of changes that happen during projects' life. In the early stages of development, commits tend to be larger because developers are adding features from scratch. As the project matures, development switches from adding new features to performing corrective changes, like bug fixes. Corrective changes are usually smaller in size.

RQ 9: How do developers communicate to others that they completed a software change?

We use the interview data to explore the activities that developers perform in order to communicate to others that they have completed a software change. We employ this complementary information to gain insights on how VCS can be improved to support development.

Table B.1 shows the qualitative codes we identified in the interview questions 2 and 4 that pertain to the activities that developers do in order to communicate that they have finished a software change.

The activities defined by the codes are not mutually exclusive. For example a developer may both use email to notify others that a change was performed and use live communication at the same time.

The common activities that developers perform consist in live communication (9/13) and email (9/13), followed by tool aided review tools (7/13). For the rest of this research question we will present these three activities in depth.

Interview participants report using email as a means of notifying others about a completed software change. Emails serve multiple functions within the development process. For example, they serve as notifications for software testers that they should begin testing a piece of changed functionality: *[I] Sent an email saying "This new functionality is available. Now test it at your convenience."* Some participants stated that in their development environment commits to VCS cause automated emails to be sent to the software testing team which uses the emails to decide what software changes to test. As one participant states, *"We move out tasks to complete and then our QA team watches our commit emails go by. So both Bazaar and Git send emails on commits to repositories. They kind of watch them go by so they can see what revisions will have the changes that they are looking for."*

Developers also use emails to notify the rest of the development team about corrective changes it needs to perform in response to that developer's change. For example, one participant mentioned an instance where he notified the development team that his database changes would cause runtime errors and also included the steps the rest of the team needed to perform in order to update to a valid database state.

Last, participants report that change completion emails may also trigger code reviews for that software change: *And then an email goes to our QA manager when that happens [a commit to VCS]. He gets an email that says "This person completed a check in. These are the files they checked in. Here's the details of what they've described they've changed". On occasion that would trigger a code review, which is a manual process for us to go and take a look and see what they've changed and what whether not we as senior staff feel that the change was appropriate.*

Similar to emails, live notification may include information on how to test the change or trigger in person code reviews. On the other hand, live communication may also take the form of weekly or daily group meetings where developers disseminate the changes they have performed: *My group has a weekly status meeting. We usually chat about things like that [finished changes].*

Some of the participants also report using tool aided review systems to announce that they completed a change. This notification is usually followed by a review of

the changes and later corrective changes to address the issues raised by the reviewers. The majority of participants that follow this approach use Git together with Github's pull request model [6] to notify the rest of the team of completed changes: developers perform changes on a separate branch. When the change is finished and ready to be merged they initiate a pull request on Github. This results in an email being automatically sent to the development team. The pull request web page allows reviewers to browse the code diffs and also to communicate with the author of the pull request. The author of the pull request may perform corrective commits in response to the reviewer's comments and append them to the pull request. When the reviewers are satisfied with the state of the pull request, they close the pull request and the changes get automatically merged into the upstream branch. One participant describes using pull requests and live communication via IRC to communicate: *and then when I make the pull request I send him the link on IRC to take a look on it, he made comments on it. And then it was mostly just me working on. He would provide maybe code samples for things on improving how code would work inside of the pull request. There really wasn't a lot of direct communication, like someone else writing and doing that, but usually it would be communicated via IRC or via the pull request.*

Observation 17: Developers predominantly use live communication and email to communicate that they completed a software change. They also make use of tool aided review systems towards this goal.

Interpretation:

This research question suggests that the activities developers use to communicate a completed change range from ad-hoc methods such as email and live communication to structured and standardized forms such as Github pull requests. Moreover, it is often the case that the ad-hoc and the structured activities coexist.

4.2.1 Implications

For researchers: We found that the average commit size slightly decreases over time. This could be explained by a variation of change practices during software development stages (development, testing, support, etc). A more detailed investigation on how software development stages influence change practices is needed.

DVCS commits contain more ITL than CVCS. This suggests that DVCS repositories are better candidates for research projects studying links between commits and issue tracking systems.

Research questions 6 and 7 show a connection between issue tracking systems and version control systems. It is common practice for software development teams to organize and break down available work using issue tracking systems. Mockus et al [42] show how Apache and Mozilla heavily depend on issue tracking systems to define, break down and keep track of work items. We propose a new avenue of research that investigates how issue management and tracking influences repository structure and commit contents.

For tool builders: As long as changes tend to become smaller as projects mature, it becomes worthwhile to rebuild smaller parts of applications as changes occur. Thus, we encourage tool builders to provide more support for intra file incremental builds.

Changes are more granular in DVCSes and usually have only one issue reference. VCS tool builders could include new abstractions that represent features. For example, cherry picking could be done at feature level.

One of the reasons why developers do not put issue numbers into commits could be the extra work it involves. Thus, tool builders could focus on a better integration between VCS and issue tracking systems.

The fact that developers use ad-hoc methods to notify others that they completed a change (observation 17) reiterates on the need for better integration between VCS and issue tracking systems: not only do they need to cross link their informational entities (e.g., commits with tasks) but they have to provide specialized notifications to interested parties: software testers need to be notified on what changes to test; software developers need to be notified on the completed changes that impact their

ongoing work; team managers need to be notified on task progress. Moreover, code change reviews were a common outcome for most interview participants upon task completion. Git and Github provide a good example on how VCS systems and online collaborative development platforms can interoperate to provide support for high level development activities such as code reviews.

For team management: Because commit size tends to become smaller as projects get older, it is reasonable to assume that developers tend to spend more time analyzing existing source code instead of adding new code. Therefore developers' productivity should be measured not only by the amount of code, but also by the complexity and importance of their changes.

We have found that participants made extensive use of email to communicate task management with their peers. We suggest that teams adopt change integration tools similar to Github's pull request feature. Such tools represent a structured means of aggregating change integration communication. They are accessible by the whole team, link changes to requirements, and leave behind a query-able artefacts in the development process.

Chapter 5: Threats to validity

Construct: Are we asking the right questions? We are interested in assessing the state of the practice for version control systems usage. Thus, we think that our research questions have high potential to provide a unique insight and value for different stakeholders: developers, researchers, tool builders and team management.

Internal: Is there something inherent to how we collect and analyze the VCS usage that could skew the accuracy of our results?

We analyzed public Git repositories and not private repositories, since private repositories are local to developers' machines and therefore inaccessible. There might be significant differences in commit contents between public and local repositories due to the history rewriting capabilities of Git. Therefore, one of the main threats is the practice of squashing commits. As we have shown in RQ 3, squashing is a used practice among software developers. For DVCS, roughly 36% of developers squash their commits. Because squashing rewrites history, it is impossible to detect squashing activity. The main effect is that commits gets larger, because squashing combines two or more commits into a single commit. The result is an increased commit size. Thus, the average commit size for DVCS might be even smaller than the ones we report. Observation 1 would still stand even in the case of heavy squashing practices.

Another threat is that our results may be biased by the development culture. As Rigby et al. [46] mention, commits done to Open Source Software (OSS) tend to be smaller than the ones done in proprietary software. However, both our SVN and Git repositories are originating from the open-source community, so the OSS culture would affect both in similar ways.

While designing our survey we aimed at keeping it short. However, in doing so, some of the participants may have misunderstood our questions. For example, when we asked the question *"Do you squash your commits?"* we were aiming to find if

developers are using the *squash* command from Git or similar tools. This command collapses together commits *after* they were committed. However, respondents might have interpreted the question as squashing multiple changes *before* committing. This can explain why 18% of developers using CVCS reported that they use squashing, even though this is not possible in CVCS tools. While we did run a pilot [24] of our survey, there is always the possibility that we have miscommunicated our intent.

One other threat is the possibility of age bias in our repositories. Since SVN has been available for a longer period of time, SVN repositories might contain older, more mature projects than Git repositories.

A possible confounding factor in repository analysis may be the project type: specific domains and architectures may affect developer behaviour on change management and thus influence commit contents. Section 3.3.1 provides a breakdown of project types present in our repository corpus. We believe our corpus is diverse enough to prevent bias towards a specific project type.

External: Are our results generalizable for the general version control usage practice?

While we analyzed 132 repositories from the open source community, we cannot guarantee that these results will be the same for proprietary (closed source) software. However, given the overall agreement between the survey, which was filled in mostly by developers working with proprietary software, and the data we acquired by analyzing open-source repositories repositories, we can assume that the general trends that we found will be true for proprietary software as well.

In our corpus of open-source repositories, 83% of the projects were developed in Java, and the remaining 16% used C/C++, Javascript and Python. Moreover, the pure Git repositories consist of 98% Java projects whereas the pure SVN repositories consist of 80% Java projects. While we have no reasons to believe that programming language affects the culture of committing changes, in the future we plan to diversify our corpus and explore change variation in programming languages.

The sources for our repositories are GitHub and SourceForge. This means

that we only looked at projects that used Git or SVN. We did not study other VCS tools for the distributed or the centralized paradigm. However, as the data from our survey indicates, Git and SVN are the predominant systems used today. They are the most widely used in their class, thus we think they are representative.

Another threat is whether we measure the effects of Git or Github. If we studied Git repositories that are not from Github, would we obtain similar results? We argue that Source Control Managers (e.g. GitHub, SourceForge, BitBucket) offer similar collaboration features such as wikis, issue tracking, pull requests (reviewing patches as a SVN equivalent), etc. Therefore, considering the similarity of features between different SCM platforms, we have reason to believe that the observed differences are due to the shift in the version control paradigm and not the platform. Further research is needed to study the effect of SCM feature variation on development.

We analyzed the top rated repositories on Github and SourceForge. We did so in order to study mature projects that represent long lived real life development efforts. Since developers have total control when choosing the changes to commit, it may be that novice programmers have different committing patterns than expert ones. This could introduce a source of bias: are our results generalizable to all repositories, or only to the top expert ones? Despite the large differences between CVCS and DVCS, we believe that future research is necessary to study how the VCS usage of novice programmers differs from that of expert ones.

Reliability: Can others replicate our results? The list of repositories we used for our analysis is available online [14]. Also, the infrastructure we used for the analysis is available open source as a GitHub repository [7].

Chapter 6: Related Work

To the best of our knowledge, our research (presented here and in our ICSE'14 paper [22]) is the first study to *compare* the impact of CVCS and DVCS on the practice of committing changes.

Several researchers [15, 16, 27, 30, 32, 33, 39, 44] studied the practice of commits but only in the CVCS paradigm. Purushothaman et al. [44] and German et al. [27] and Hindle et al. [32] studied the properties of typical small commits or typical large commits. Hattori et al. [30] study the size of commits with the purpose of classifying changes. Hofmann et al. [33] predict commit size based on commit history. Herzig et al. [31] propose an algorithm for untangling changes in CVCS. Arafat et al. [16] and Alali et al. [15] studied the distribution of commit size in SVN repositories. However, ours is the first study to compare the commit size in CVCS and DVCS. The SVN commit size reported by related work is similar in range to ours [15, 16].

Related with our study about the impact and the presence of issue tracking systems (ITS), several researchers [17, 20, 29, 40, 51] studied ITS. Tian et al. [51] employ machine learning to infer bug fixing commits. Bird et al. [20] and Bachmann et al. [17] present tools and approaches to link source code, ITS, and mailing archives. Meneely et al. [40] makes suggestions on improving issue tracking labeling in commit messages. Hassan et al. [29] provide an overview of repository and ITS mining practices. However, none of these studies compared CVCS and DVCS based on ITS practices.

Recently, researchers have started mining DVCS repositories. A body of work [21, 28, 36] studies the technical challenges of analyzing Git repositories and mining Github.

Multiple researchers [18, 23, 45, 48, 52–54] studied the shape of Git repositories and the Github community. Biazzini et al. [18] explored recurring commit topologies in DVCS while Yu et al. [54] mined social network topologies on Github. Tsay et al. [52] and Sheoran et al. [48] studied social aspects of Github such as pull request

discussions or user roles. Ray et al. [45], Vasilescu et al. [53], and Casalnuovo et al. [23] used Github to study aspects of programming languages and development processes. In contrast to these studies we focus on how version control paradigms influence software changes.

Other research is focusing on how DVCS are affecting development [35, 43, 47]. Muslu et al. [43] explored the transition to DVCS in a large company together with barriers and outcomes. Kalliamvakou et al. [35] and Rigby et al. [47] study the effects of DVCS on the organization of open source and commercial projects. These studies complement our own. While they look at how DVCS and DVCS collaboration platforms affect the organization and development process of teams and companies, our study focuses on how DVCS affect software changes.

Chapter 7: Conclusions

In this paper we present the first in-depth study to measure the impact of DVCS on software change. To this end we ran a survey with 820 participants, analyzed a corpus of 132 repositories, and interviewed 13 developers.

We found that the use of CVCS and DVCS have observable effects on developers, teams, and processes. The most surprising findings are that (i) the size of commits in DVCS was smaller than in CVCS, (ii) developers split commits (group changes by intent) more often in DVCS, and (iii) DVCS commits are more likely to reference issue tracking labels. These show that DVCS contain higher quality commits compared to CVCS due to their smaller size, cohesive changes and the presence of issue tracking labels. Thus, in our sample of data, DVCS adopters are taking advantage of the paradigm's empowering capabilities to improve the structure of their commits.

The survey provided valuable information on why developers prefer one paradigm versus the other. DVCS are preferred because of *killer features*, such as the ability of committing locally. In contrast CVCS are preferred for their ease of use and faster learning curve.

Through the interviews we learned that developers would prefer the best of both worlds when committing: they like small commits for ease of searching, manipulating, and understanding past changes but they also like large feature commits that keep history concise and group together related changes.

We hope that our work inspires future research not only into the impact that centralized and distributed VCS tools have on software development but also on how general properties of VCS tools enable developers to manage and express change.

Bibliography

[1] Agilepdx: a local software developer conference in portland. http://agilepdx.org.

[2] Cvs. http://cvs.nongnu.org/. Accessed February 27, 2014.

[3] Git. http://git-scm.com/. Accessed February 27, 2014.

[4] gitective: Git repository analysis tool. https://github.com/kevinsawicki/gitective. Accessed September 6, 2013.

[5] Github. http://www.github.com/. Accessed February 27, 2014.

[6] Github flow: the pull request model. https://guides.github.com/introduction/flow/index.html.

[7] http:/www.github.com/caiusb/gitsvn.

[8] jgit: An implementation of the git version control system in pure java. https://github.com/eclipse/jgit. Accessed September 6, 2013.

[9] Mercurial. Accessed February 27, 2014.

[10] Sourceforge. http://www.sourceforge.net/. Accessed February 27, 2014.

[11] Sourceforge research data archive (srda): A repository of floss research data. http://srda.cse.nd.edu/mediawiki/index.php?title=Main_Page. Accessed September 6, 2013.

[12] Svn. http://subversion.tigris.org/. Accessed February 27, 2014.

[13] svn2git: Svn to git repository conversion tool. https://github.com/nirvdrum/svn2git. Accessed September 6, 2013.

[14] Vcs usage study companion. http://cope.eecs.oregonstate.edu/VCStudy/.

[15] Abdulkareem Alali, Huzefa Kagdi, and Jonathan I Maletic. What's a typical commit? a characterization of open source software repositories. In *Program Comprehension, 2008. ICPC 2008. The 16th IEEE International Conference on*, pages 182–191. IEEE, 2008.

[16] Oliver Arafat and Dirk Riehle. The commit size distribution of open source software. In *System Sciences, 2009. HICSS'09. 42nd Hawaii International Conference on*, pages 1–8. IEEE, 2009.

[17] Adrian Bachmann and Abraham Bernstein. Data retrieval, processing and linking for software process data analysis. *University of Zurich, Technical Report*, 2009.

[18] Marco Biazzini, Martin Monperrus, and Benoit Baudry. On analyzing the topology of commit histories in decentralized version control systems. In *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on*, pages 261–270. IEEE, 2014.

[19] Christian Bird, Adrian Bachmann, Eirik Aune, John Duffy, Abraham Bernstein, Vladimir Filkov, and Premkumar Devanbu. Fair and balanced?: bias in bug-fix datasets. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 121–130. ACM, 2009.

[20] Christian Bird, Adrian Bachmann, Foyzur Rahman, and Abraham Bernstein. Linkster: enabling efficient manual inspection and annotation of mined data. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, pages 369–370. ACM, 2010.

[21] Christian Bird, Peter C Rigby, Earl T Barr, David J Hamilton, Daniel M German, and Prem Devanbu. The promises and perils of mining git. In *Mining Software Repositories, 2009. MSR'09. 6th IEEE International Working Conference on*, pages 1–10. IEEE, 2009.

[22] Caius Brindescu, Mihai Codoban, Sergii Shmarkatiuk, and Danny Dig. How do centralized and distributed version control systems impact software changes? In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 322–333, New York, NY, USA, 2014. ACM.

[23] Casey Casalnuovo, Prem Devanbu, Abilio Oliveira, Vladimir Filkov, and Baishakhi Ray. Assert use in github projects. 2015.

[24] Daniela S Cruzes and Tore Dyba. Recommended steps for thematic synthesis in software engineering. In *Empirical Software Engineering and Measurement (ESEM), 2011 International Symposium on*, pages 275–284. IEEE, 2011.

[25] Martín Dias, Alberto Bacchelli, Georgios Gousios, Damien Cassou, and Stéphane Ducasse. Untangling fine-grained code changes.

[26] Klint Finley. Github has surpassed sourceforge and google code in popularity. `https://www.readwriteweb.com/hack/2011/06/github-has-passed-sourceforge.php`, 2011. Accessed February 27, 2014.

[27] Daniel M German. An empirical study of fine-grained software modifications. *Empirical Software Engineering*, 11(3):369–393, 2006.

[28] Georgios Gousios and Diomidis Spinellis. Ghtorrent: Github's data from a firehose. In *Mining Software Repositories (MSR), 2012 9th IEEE Working Conference on*, pages 12–21. IEEE, 2012.

[29] Ahmed E Hassan. The road ahead for mining software repositories. In *Frontiers of Software Maintenance, 2008. FoSM 2008.*, pages 48–57. IEEE, 2008.

[30] Lile P Hattori and Michele Lanza. On the nature of commits. In *Automated Software Engineering-Workshops, 2008. ASE Workshops 2008. 23rd IEEE/ACM International Conference on*, pages 63–71, 2008.

[31] Kim Herzig and Andreas Zeller. The impact of tangled code changes. In *Proceedings of the Tenth International Workshop on Mining Software Repositories*, pages 121–130. IEEE Press, 2013.

[32] Abram Hindle, Daniel M German, and Ric Holt. What do large commits tell us?: a taxonomical study of large commits. In *Proceedings of the 2008 international working conference on Mining software repositories*, pages 99–108. ACM, 2008.

[33] Philipp Hofmann and Dirk Riehle. Estimating commit sizes efficiently. In *Open Source Ecosystems: Diverse Communities Interacting*, pages 105–115. Springer, 2009.

[34] Jiří Janák. Issue tracking systems. Master's thesis, Masaryk University, Brno, 2009.

[35] Eirini Kalliamvakou, Daniela Damian, Kelly Blincoe, Leif Singer, and D German. Open source-style collaborative development practices in commercial projects using github. In *Proceedings of the 37th International Conference on Software Engineering (ICSE?15)*, 2015.

[36] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M. German, and Daniela Damian. The promises and perils of mining github. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 92–101, New York, NY, USA, 2014. ACM.

[37] Michele Lanza and Radu Marinescu. *Object-oriented metrics in practice: using software metrics to characterize, evaluate, and improve the design of object-oriented systems*. Springer-Verlag New York Inc, 2006.

[38] Radu Marinescu. Detection strategies: Metrics-based rules for detecting design flaws. In *Software Maintenance, 2004. Proceedings. 20th IEEE International Conference on*, pages 350–359. IEEE, 2004.

[39] Maryam Marzban, Zahra Khoshmanesh, and Ashkan Sami. Cohesion between size of commit and type of commit. In *Computer Science and Convergence*, pages 231–239. Springer, 2012.

[40] Andrew Meneely, Mackenzie Corcoran, and Laurie Williams. Improving developer activity metrics with issue tracking annotations. In *Proceedings of the 2010 ICSE Workshop on Emerging Trends in Software Metrics*, pages 75–80. ACM, 2010.

[41] Tom Mens. A state-of-the-art survey on software merging. *Software Engineering, IEEE Transactions on*, 28(5):449–462, 2002.

[42] Audris Mockus, Roy T Fielding, and James D Herbsleb. Two case studies of open source software development: Apache and mozilla. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(3):309–346, 2002.

[43] Kivanc Muslu, Christian Bird, Nachiappan Nagappan, and Jacek Czerwonka. Transition from centralized to distributed vcs: A microsoft case study on reasons, barriers, and outcomes. In *ICSE '14: Proceedings of the 2012 International Conference on Software Engineering*. IEEE Press, 2014.

[44] Ranjith Purushothaman and Dewayne E Perry. Toward understanding the rhetoric of small source code changes. *Software Engineering, IEEE Transactions on*, 31(6):511–526, 2005.

[45] Baishakhi Ray, Daryl Posnett, Vladimir Filkov, and Premkumar Devanbu. A large scale study of programming languages and code quality in github. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 155–165. ACM, 2014.

[46] P.C. Rigby, B. Cleary, F. Painchaud, M. Storey, and D.M. German. Contemporary peer review in action: Lessons from open source development. *Software, IEEE*, 29(6):56 –61, nov.-dec. 2012.

[47] Peter C Rigby, Earl T Barr, Christian Bird, Prem Devanbu, and Daniel M German. What effect does distributed version control have on oss project organization? In *Release Engineering (RELENG), 2013 1st International Workshop on*, pages 29–32. IEEE, 2013.

[48] Jyoti Sheoran, Kelly Blincoe, Eirini Kalliamvakou, Daniela Damian, and Jordan Ell. Understanding watchers on github. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 336–339. ACM, 2014.

[49] Eric Sink. *Version control by example*. Pyrenean Gold Press, 2011.

[50] Eric Sink. *Version control by example*. Pyrenean Gold Press, 2011.

[51] Yuan Tian, Julia Lawall, and David Lo. Identifying linux bug fixing patches. In *Software Engineering (ICSE), 2012 34th International Conference on*, pages 386–396. IEEE, 2012.

[52] Jason Tsay, Laura Dabbish, and James Herbsleb. Let's talk about it: evaluating contributions through discussion in github. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 144–154. ACM, 2014.

[53] Bogdan Vasilescu, Stef van Schuylenburg, Jules Wulms, Alexander Serebrenik, and Mark GJ van den Brand. Continuous integration in a social-coding world: Empirical evidence from github. In *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on*, pages 401–405. IEEE, 2014.

[54] Yue Yu, Gang Yin, Huaimin Wang, and Tao Wang. Exploring the patterns of social behavior in github. In *Proceedings of the 1st International Workshop on Crowd-based Software Development Methods and Technologies*, pages 31–36. ACM, 2014.

[55] Andreas Zeller. Yesterday, my program worked. today, it does not. why? In *Software Engineering-ESEC/FSE'99*, pages 253–267. Springer, 1999.

APPENDIX

Appendix A: Survey and interview questions

Table A.1: Interview Questions.

Q1.	Can you please describe how you recorded a recent and completed software change that you did?
Q2.	How did you communicate to others that the change was made?
Q3.	Can you please describe how you recorded an older software change that you did?
Q4.	How did you communicate to others that the change was made?

Table A.2: Survey questions. Open-ended answers are marked with an asterisk.

Demographic	
Q1.	What is the type of project that you spend most of your time on?
Q2.	What is the extent of your programming experience?
Q3.	How old is the project that you typically work on?
Q4.	What type of VCS paradigm do you prefer?
Q5.	What is the VCS tool that you use most often?
Q6.	What is the size of your software development team?
Commit practice	
Q7.	How often do you commit your changes?
Q8.	If you chose "It depends", what does it depend upon? *
Q9.	When you commit, how do you group your changes?
Q10.	If applicable, what criteria do you use to split your commits? *
Q11.	If applicable, do you squash your commits?
Q12.	If you squash your commits, what are your reasons for squashing? *
Q13.	Which of the following VCS do you find the most natural to commit changes?
Q14.	Why do you find it most natural? *
Issue tracking	
Q15.	Does the project where you sped most of your time have a commit policy?
Q16.	Do you use an issue tracking system?
Q17.	If yes, do you work on more than one issue at a time?
Q18.	If yes, are the issues related?
Q19.	If you work on several issues at once, do you commit each issue separately?
Q20.	How often do you share changes with other members?

Appendix B: Qualitative codes

Table B.1: Codes extracted from interview questions Q2 and Q4

Code	Definition	Example
Email	Send email to notify others of completed a change	*"I sent out email to let people know that the fix had been completed."*
No communication	Do not communicate completion of change	*"I committed it and he could see that I committed stuff but I did not specifically go over and tell him that this is done."*
Live communication	Communicate completion of change in person or via live messaging	*"We're collocated so I'll just get up from my cube, walk around and sort of go over what they think the testing needs to be."*
Tool aided review	Submit the completed change to a code review platform	*"Coordination happens over Github. They would review my change."*
Corrective changes	Perform corrective changes in response to a review	*"they suggested some changes. I made those in subsequent commits."*
Mark in ITS	Mark the change as completed in an Issue Tracking System	*"We move out tasks to complete."*
Release notes	Write release notes for the completed change	*"I would write up release notes."*

Table B.2: Codes extracted from the survey open ended question Q10

Code	Definition	Example
Fine-grained scope	Splitting changes by having a finer grained scope in mind (e.g., changes at method level, refactorings, small fixes)	*"Small, logical chunks, complete thoughts"*
Coarse-grained scope	Splitting changes by having a coarser scope in mind (e.g., larger functionality, bug-fixes)	*"The changes and working tests around a specific piece of functionality."*
Policy	A criteria that is imposed from the outside (management, development process etc.)	*"The commits are designed to be easy to review individually, and to allow individual reversion of semantically discrete change-sets."*
Other	Reasons that do not fit in the above criteria	*"No criterion. It's just random."*

Table B.3: Additional Codes that refer to splitting commits extracted from interview questions Q1 and Q2

Code	Definition	Example
Subtask	Try to break large units of work into subtasks and then commit the subtasks	*"We try to compartmentalize the feature updates as small as possible"*
Handoff Commit	Commit in order to handoff the changes to another developer	*"I had something that wasn't compiling that I needed to share with another developer"*
Conflicting granularity	There are different requirements on commit rules	*"there are some people that feel that everything that is committed should compile and have running tests all the time. But that kinda limits how often you commit things"*
Code Dump	Commit changes to the VCS with no specific strategy	*"I commit once a day"*

Table B.4: Codes extracted from the survey open ended question Q12

Code	Definition	Example
Group Similar Changes	Squashing with the intent of having only one commit per logical change (feature, bug-fix etc)	*"Usually to combine a number of incremental work in progress commits into one coherent one."*
Removing irrelevant intermediate steps	Removing commits after a feature was done, because they are no longer relevant	*"Get rid of intermediate commits that don't represent logical application development."*
Removing mistakes	Merging two or more commits: one that introduced an error and the others that fix it.	*"Fixing mistakes or oversights from a previous commit that hasn't been pushed yet; adding changes that should logically be part of that commit"*
Keeping history clean	Reducing the clutter in the main branch or repository	*"It keeps our master repo clean, as there is a large number of contributors."*
Policy	Requirement coming from the outside (management, development process etc)	*"I push as one commit to the projects development branch. This approach is imposed"*
Other	Answers that do not fit in any of the above criteria	*"It's applicable."*

Table B.5: Codes extracted from the survey open ended question Q14

Code	Definition	Example
Killer feature	Presence of a certain feature that makes life easier	*"The merging, branching, and rebasing functionality are extremely fast and convenient."*
Old habit	The only one they have used or just simple habit	*"The only one I've used"*
Easy to use	Integration with other tools, perceived simplicity compared to other tools	*"Very easy and few steps / commands. Easy to understand what's going on."*
Personal preference	They just prefer one over the other	*"Just seemed to make sense"*
Other	Answers that do not fit in the above buckets	*"Can't specify, just feels natural."*