How Do Centralized and Distributed Version Control Systems Impact Software Changes?

Caius Brindescu, Mihai Codoban, Sergii Shmarkatiuk, Danny Dig
EECS School at Oregon State University
{brindesc,codobanm,shmarkas,digd}@eecs.oregonstate.edu

ABSTRACT

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 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. 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, and (iii) DVCS commits are more likely to have references to issue tracking labels.

1. INTRODUCTION

Distributed Version Control Systems (DVCS) like Git [2] or Mercurial [6] are widely used today. Over the last couple of years GitHub [4], which is the most popular repository hosting service for Git projects, has taken the open source community by storm. At the end of 2012, GitHub hosted over 4.6M repositories. Compare this with the previous paradigm, CVCS, epitomized by SVN [9] and CVS [1]. SourceForge [7], the primary repository hosting service for SVN had about 300K repositories by the end of 2012. Also, our own survey of 820 developers shows that 65% use DVCS and 35% use CVCS.

DVCS brings a whole set of novel capabilities. Using DVCS, developers (i) can work in isolation on local copies of the repositories enabling them to work offline while still retaining full project history, (ii) they can cheaply create and merge branches, and (iii) they can commit individual changed lines in a file, as opposed to being forced to commit a whole file like in CVCS.

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 to invest time and effort to transition to these new tools. Also, researchers are in danger of making errors when mining repositories, due to confounding effects imposed by DVCS. Finally, tool builders can build the wrong tools if they are not aware of developers' habits.

In this paper we present the first large-scale study that answers in-depth questions about the extent to which DVCS influences the practice of managing changes. 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.

To get further insights into how DVCS affects code changes, also we analyzed 409M lines of code changes from 358300 commits, 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 switched to the distributed paradigm. The hybrid repositories can reveal whether 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.

Using the data from our survey and our mining of repositories, we answer 12 research questions organized in three overarching themes:

Theme 1: How does the VCS type 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.

Theme 2: How does the team size affect VCS usage?

RQ 6: *Does team size affect the choice of VCS?*

RQ 7: *Are larger teams more likely to use Issue Tracking Systems (ITS)?*

RQ 8: *Does team size influence commit squashing?*

Teams of all sizes prefer using DVCS. The team size does not influence the size of commits. Most teams include issue tracking labels in their commits.

Theme 3: How does the VCS type affect the development process?

RQ 10: *Does the type of VCS influence the presence and the number of issue tracking labels (ITL)?*

RQ 11: *Is there a correlation between the number of issue tracking labels (ITL) in the commit message and the commit size?*

RQ 12: *How does the size of commits vary in time?*

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/precise software changes (e.g., bug prediction, inferring high-level meaning for code changes, etc.) they should mine distributed repositories. *Developers* can give more precise meaning to their changes when they use distributed version control systems. *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 12 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 a set of tools to collect metrics and analyze centralized and distributed repositories. We applied these tools on a corpus of 132 repositories.
4. **Implications.** We present implications of our findings from the perspective of four different audiences: *researchers*, *developers*, *tool builders*, and *team managers*.

The tools, summary of survey responses, and corpus are publicly available at: `http://tiny.cc/VCStudy`

2. EXPERIMENTAL SETUP

In this section we describe the two sources of data we used to answer our research questions.

2.1 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 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%).

2.1.1 Classification of open-ended questions

The survey contained both multiple choice and open-ended questions [1]. We hand-coded the answers to the open-ended questions using *qualitative thematic coding* [18]. 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 first and the third authors, developed the categories which were not known apriori. For measuring the agreement we used the Jaccard coefficient.

[1]Survey contents can be accessed at `http://goo.gl/9eSH4W`

Table 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%

The categories that emerged from coding can be seen as rows in Tables 12, 9, and 11.

Tables 3, 4, 5 show more details (i.e., criteria, definitions, and examples) that we used for classifying the open ended questions regarding commit splitting, squashing and VCS usability, respectively.

2.2 Repository

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

2.2.1 Repository Corpus

In order to answer our research questions we needed to collect 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 paradigm. Our assumption is that differences in metrics taken from these three kinds of repositories provide valuable insights on how they influence source code management.

We collected SVN repositories from SourceForge as representing the centralized paradigm, and Git repositories from GitHub as representing the distributed paradigm. These repositories span several programming languages: Java, C, C++, and JavaScript.

For GitHub we selected the top ranked repositories, i.e.,

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?

Table 1: Survey questions. Open-ended answers are marked with an asterisk.

Code	Definition	Example
Fine-grain 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"*
Course-grain 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 3: Classification of reasons for splitting commits

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 [8], 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 select hybrid repositories, we searched for internet posts about migrating repositories from SVN to Git. In addition, while collecting Git repositories, some of them proved to have actually started in SVN. Thus we classified them as hybrid. We distinguish the two stages of hybrid repositories as Hybrid$_{SVNStage}$ and Hybrid$_{GitStage}$.

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.

Table 6 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.

Table 6: 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

2.2.2 Repository Analysis

We have built an analysis platform to gather several commit metrics. We used Git as the canonical representation for all repositories. This is possible since the Git object storage

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 4: Classification of reasons for squashing commits

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."*

Table 5: Classification of reasons why developers find a VCS natural to use

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 [10].

Our analysis platform builds on top of GITECTIVE [3], a framework capable of traversing history trees, one commit at a time.

In order to explore the statistical significance of various sample differences, we applied the Wilcoxon rank-sum test. We chose this test since none of the data fit a normal distribution.

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

Filtering changes.

In our initial manual investigation of commits we have discovered that many commits do not represent actual programming changes carried out by a developer (e.g., adding 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. Our analysis filters 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.
- 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 manufactured by repository migration tools.

In addition, inside each commit, our analysis discards lines of code that (i) introduce or modify code comments, and (ii) only modify white space (e.g., code formatting).

Commit metrics.

For each commit we collect the following metrics.

Commit id, useful for identifying commits.

Commit date, useful for sorting commits chronologically.

The author of the commit, useful in grouping data by authors.

Number of LOC changed by the commit, useful 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, useful for determining the commit size. While LOC 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, useful 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 [26], etc. In order to detect them, we used an approach similar to the one described by Bird at al. [15], which employs searching for specific text patterns in the commit message.

3. RESULTS

3.1 How does the VCS type affect developers' behavior?

RQ 1: Does the type of VCS affect the size of commits?
Table 7 shows the commit size, both in lines of code and in number of files, made by individual authors. This data is grouped by VCS type.

Table 7: Commit size across different VCS

	Mean		Median		StdDev	
	LOC	files	LOC	files	LOC	files
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
HybridGit	23.02	2.40	11.52	1.70	27.57	1.74
HybridSVN	25.72	2.82	12.61	1.96	31.24	2.15

In terms of LOC, the commits from Git repositories tend to be smaller than those made in SVN repositories ($p < 0.01$). The mean and median lines of code changed per commit in Git repositories is 27.20 and 13.46, respectively, while for SVN repositories these values are 40 and 18.44 respectively. The standard deviation of changed lines of code also differs, with 32.72 lines of code for Git and 49.62 lines of code for SVN.

In terms of the number of files that are affected by a commit, the same trend of a smaller commit size can be seen as with the commit lines of code, although at a lower significance level ($p = 0.2$). Commits from Git repositories tend to affect fewer files than commits from SVN repositories.

On the other hand, hybrid repositories do not show a smaller commit size after they transition to Git ($p>0.5$).

Observation 1: DVCS repositories have a smaller commit size than CVCS repositories, both in terms of lines of code and impacted files.

Observation 2: Hybrid repositories do not show any difference between the size of commits performed before and after the switch to the distributed paradigm.

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 local 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""*. Also, resolving conflicts becomes a task that is consciously entered into by the user only when she synchronizes her changes with other team members. It is not something that happens haphazardly with every local commit.

Hybrid repositories on the other hand do not seem to experience smaller commits after switching to Git, as observation 2 shows. Our assumption is that in such cases a certain commit policy is formed within the team while the project is under the SVN version control stage. This commit policy is then intuitively carried over in the Git version control stage, leading to the same manner of composing software change as before.

The culture of the project takes a longer 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 during an extended coding session might belong to one or more logical units of change. Do developers bother to split these changes and commit separately? Or do they just group their changes by committing everything that was modified in one large commit? The answers in the survey give us the picture depicted in Table 8.

Table 8: 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%).

One explanation for this fact 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.

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, modify loop termination condition, 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 9: Reasons for splitting commits (%)

Technique	DVCS	CVCS	Overall
Implementation details	37.01	21.85	32.03
Intent of change	45.13	62.251	50.76
Policy	6.17	5.30	5.88
Other	11.69	10.60	11.33

Table 9 tabulates the reasons for splitting commits.

We observe that in the case of DVCS, developers chose to split their changes based on implementation details more frequently than they do in CVCS. As is the case with observation 3, we can attribute this to an easier commit process. Splitting commits by implementation details will inevitably result in more commits. Being easier and cheaper to commit may make this process more attractive to developers.

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 in the repositories 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.

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

Squashing refers to the operation of merging two or more commits into a single one.

Results from the survey show that only 30% of the developers squash their commits. If we separate the data into two bins, one for distributed and one for centralized repositories we get the results in Table 10.

Table 10: 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 10 shows that squashing happens twice more often in distributed repositories than in centralized ones[2]. 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 11): (i) they do so to group several changes together and; (ii) they do not care about the path they took to a solution as long as it's finished and it works.

[2]See Internal Threats to Validity (Section 4)

Table 11: 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

Observation 6: 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 become accustomed 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 12 summarizes the complete results.

Table 12: 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.

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

Observation 7: The commit process of DVCS is perceived by developers to be more natural because of the presence of killer features.

Observation 8: The commit process of CVCS is perceived to be more natural because of familiarity and a faster learning curve, not their feature set.

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

Table 13 shows results we obtained from the survey. Developers commit 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%).

Observation 9: Most developers have similar habits independent of what VCS they use.

Table 13: 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.

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 dbeen added. Instead of going through thousands of changes, the reviewer can go through a natural progression of well defined changes that slowly introduce the new functionality, one discrete change at a time.

Also, smaller commits enables easier bisecting. This enables techniques such as Delta Debugging [36] to be employed to find the root cause of bugs.

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

Switching to the new systems and paradigms can help developers structure and understand source code changes better. 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. This way, they can make it very likely for a random commit to build and run or pass the tests. 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 6 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 where some changes come from.

From observation 7 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 contour individual changes. For researchers who tie different software artifacts to code, such as bugs, Git commits are much more surgical therefore they may have fewer false mappings.

Researchers must be careful when collecting software repository related metrics. We have found that old repositories that migrated through several VCS tools present a different behavior than pure repositories. It may be the case that the culture formed in the era of the first VCS shadows the subsequent ones. There might other phenomena that influence a repository's structure. The type of issue tracking system, the width and height of the forking tree or the branching model could be some of them. 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 2.2.2, 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 other. 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 poses 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. When conducting research on such repositories, this threat must be taken into account.

For tool builders: Although Git enables finer grained changes, it still falls upon the developers' shoulders to disentangle these changes. This is a manual, tedious, and time consuming task. VCS tools could keep track of different change intents and then offer to commit them separately. Herzig et al. [23] show a technique by which this can be achieved. They devised a heuristic untangling algorithm that splits tangled changes according to different source code criteria (e.g., the distance between two changed AST nodes).

We envision a new generation of tools that can use the average value of commit size as a quality metric. When a developer has uncommitted code larger than a quality 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 [27]. Marinescu [28] 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.

Squashing is a process by which history can be altered and / or completely lost. In order 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. Thus, instead of loosing history through squashing, developers could group commits into larger, composite commits.

Our finding that developers from hybrid repositories use the same habits after switching to DVCS as when they used CVCS suggests the need for tools to help educate developers on how to effectively change their habits.

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 do not show the same trends as non hybrid ones. 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.

3.2 How does the team size affect VCS usage?

RQ 6: Does team size affect the choice of VCS?

Table 14: VCS choice by team size (%)

VCS type	1	2-5	6 - 10	11 - 25	26 - 100	> 100
DVCS	82.22	72.07	62.30	65.22	60.00	70.97
CVCS	17.78	27.93	37.70	34.78	40.00	29.03

Table 14 shows that most teams use the distributed model, regardless of the team size. However, the presence of the centralized version control systems increases once the team size increases.

Interpretation: Since 53% of the survey projects started less than two years ago, it is likely that they were developed during the rising popularity of the DVCS. As for the popularity of CVCS at larger teams, this can be interpreted as inertia of larger teams to use new paradigms and tools. However, once the team size crosses 100, the overhead of merging changes pushes the team against their inertia.

Observation 10: Teams of all sizes predominantly prefer DVCS

RQ 7: Are larger teams more likely to use Issue Tracking Systems (ITS)?

We are answering this question using two data sources, the survey and the analysis of the repositories.

Table 15 summarizes results collected from the survey.

Table 15: Issue Tracking System (ITS) usage based on team size (%)

Team size	Use an ITS	Don't use an ITS
1	63.04	36.96
2 - 5	97.88	2.12
6 - 10	92.23	7.77
10 - 25	95.24	4.76
26 - 100	97.01	2.99
Over 100	100.00	0.00
Overall	93.87	6.13

Overall, 93.87% of the participants reported using an ITS. While the value is lower for one-person projects (63%), it is over 90% in all other cases. More interestingly, all participants working in projects with over 100 developers report using an ITS.

The analysis of the repositories shows that 91.6% of the projects contain commit messages that refer to issues in an ITS. This correlates very closely with the data we have obtained from the survey.

Observation 11: Most projects use an Issue Tracking System.

RQ 8: Does team size influence commit squashing?

Another question that we ask is whether developers working in larger team squash more frequently than developers working in smaller teams. Table 16 shows that this could be

Table 16: Squashing in relation to the team size (%)

Team size	Squash	Don't squash	Not Applicable
1	17.39	56.52	26.09
2-5	27.27	54.55	18.18
6-10	42.25	29.50	28.06
11-25	30.16	50.79	19.05
26-100	40.00	46.15	13.85
Over 100	57.14	34.29	8.57

the case. While teams of two to five developers squash only 27% of the time, teams with over 100 developers squash 57% of the time. This is more than double the rate compared to smaller teams. The reasons for this are twofold:

1. According to Observation 10, larger teams tend to use CVCS more often.
2. Developers might do this in order to keep the upstream history clean. Working in a larger team means that if every developer were to push their full history, the main repository could get too messy. So squashing would be a way to mitigate an explosion in the number of commits and branches.

Observation 12: Large teams squash commits more often.

RQ 9: Does team size affect the size of commits?

We measured the correlation coefficient between team size and commit size (measured in lines of code and number of changed files). Table 17 shows that there is no linear relation between team size and the size of commits. This holds for both LOC and number of files. The fact that all but one coefficient are negative could indicate a weak downhill trend for commit size as team size increases.

Table 17: Correlation coefficients between team size and commit size

Repository type	LOC	# of files
Git	-0.11	-0.09
SVN	-0.07	-0.16
Hybrid$_{GitStage}$	-0.01	0.16
Hybrid$_{SVNStage}$	-0.06	-0.35

Observation 13: There is no discernible relationship between the team size and the size of commits other than a weak tendency for commit size to decrease as team size increases.

Interpretation: We initially expected to see that commit size decreases as team size increases. Small teams can be very agile and quickly grow the code base because everybody knows what everybody else is doing. As team size increases developers have less knowledge of the overall task distribution, and they must exercise more care when accepting and integrating changes from many sources. Therefore, we assumed that large teams should perform smaller commits to better express changes.

The need for extra care about management of software changes would require more clear commits that address cohesive and understandable changes. This is achieved by splitting large commits into smaller commits that tackle only one intent of change.

However, data shows that developers tend to use the same intuition for splitting changes, regardless of team size. The lack of a clear relationship between team size and commit size suggests that teams do not use the size of commits as an implicit way of simplifying understanding and management of software changes. This hints that large teams must use other mechanisms to control the complexity of software changes.

Such mechanisms could be more complex branching and merging models. Philips et al. [31] show that as projects grow in size and activity, the complexity of the branching model increases.

3.2.1 *Implications*

For researchers: Best quality data about issue tracking systems is obtained from projects developed by large teams.

On the other hand, large teams tend to squash more often which would results in bigger commits with more entangled changes. Therefore, researchers may have to trade off some traits over other ones when choosing to study repositories from small or large teams.

For tool builders: Practically all large teams use Issue Tracking Systems in order to track work items. However, in the current state of practice developers track code and issues by inserting issue references inside commit messages. This is tedious and imprecise since developers have to manually group changes by issue.

Therefore, tool builders should create tools that (i) keep track of code written for a particular task, (ii) automatically group code changes by issue, and (iii) incorporate issue tracking inside Version Control Systems.

Tool builders should abstract away low level VCS concepts such as branches or revisions in order to ease the learning curve of Version Control Systems. For example, a developer could tell the ITS that he wants to work on a particular task. The ITS would then silently create a branch in the background for that particular task. When the developer tells the ITS that he wants to work on another feature, the VCS would invisibly switch branches. When the developer finishes a task, he could mark it as ready. This would silently trigger a pull request to the gatekeeper. If the gatekeeper is happy with the feature implementation, he tells the ITS to accept the feature. This would silently trigger a merge. In this manner, developers work with high level development process entities such as features rather than with low level tool implementation details such as branches.

For team management: Very large teams that struggle with aggregating changes should consider investing the extra effort and switch to distributed tools (Git, Hg, etc) and more involved branching models (specialized branches, deeper forking tree, etc) [31].

3.3 How does the VCS type affect the development process?

RQ 10: Does the type of VCS influence the presence and the number of issue tracking labels (ITL)?

From the survey, we found that the majority of developers (69%) commit only one issue at a time. The figure is slightly smaller for CVCS than it is for DVCS. As survey results show, the percentage of developers who commit more than

Table 18: 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

one issue at a time is 17% if they use CVCS and 9% if they use DVCS. Data from analysed repositories show that 9.2 % of CVCS commits with ITL and 4.4% of DVCS commits with ITL contain more than one issue tracking label.

Table 19: Repository Analysis: Distribution of commits with one ITL and with many ITLs

	1 issue	>1 issues
SVN	96.13	3.87
Git	98.04	1.96
Hybrid	97.04	2.96
Overall	97.37	2.63

Repository analysis data shows that 13.13% of commits to CVCS, 43.42% of commits to DVCS and 33.12% of commits to Hybrid repositories contain ITL. Overall percentage of commits with ITL without regard to VCS type is 30.95%.

Observation 14: A small number of commits are labelled with ITL. Nevertheless, issue labels appear more frequently in DVCS commit messages than in CVCS commit messages.

The fact that we see a larger 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. As mentioned before, in most DVCS (such as Git and Mercurial) the user is able to commit at the line level. This difference to the granularity of change selection could explain the results.

From analysing the repositories, we found that only 30.95% of commits have an ITL in their commit message. This mirrors similar findings by Bird et al. [15]. 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%.

These higher values in Git repositories, compared to SVN repositories, show that developers assign issue references to commits in Git more frequently than they do in SVN. This supports the claim that developers find it easier to assign issue labels in DVCS.

The low overall number of commits with ITL references can be attributed to a relaxed committing policy. As we mentioned in section 2, the repositories are gathered from open source projects. It may be difficult to enforce a strict commit policy for such repositories.

RQ 11: Is there a correlation between the number of issue tracking labels (ITL) in the commit message and the commit size?

Table 20 shows a strong correlation for SVN and Hybrid repositories (0.68 and 0.81 correspondingly), and a weak negative correlation for Git repositories. The overall correlation was computed by taking into consideration all repositories, regardless of their VCS type.

We can see that commits to SVN and Hybrid repositories tend to be larger when more issues appear in the commit message. For Git repositories this trend does not hold. Even more, there is a slight tendency for commit size to decrease when the number of issue tracking labels increases.

Table 20: Average LOC 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 15: In SVN and hybrid repositories commit size is positively correlated with the number of referenced issues.

Observation 16: In Git repositories there is a weak trend for commit size to decrease when the number of referenced issues increases.

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.

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, the commit could contain a common piece of code that addresses all referenced issues.

Observations 3 and 5 show that developers using DVCS split their commits more often and 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. By doing so, developers treat feature interaction more elegantly and also send out a cleaner message of their intent.

RQ 12: How does the size of commits vary in time?

In order to investigate how 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 21 shows that commit size tends to become smaller as projects get older. The average age of a typical repository (time between first and 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.

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. 71% 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%.

Table 21: Correlation between commit size and commit time

VCS	average correlation	number of positive correlations	number of negative correlations	percent of negative correlations
SVN	-0.06	21	31	60%
GIT	-0.17	13	25	66%
Hybrid	-0.11	12	16	57%
ALL	-0.11	46	72	61%

Observation 17: Commit size tends to become smaller as projects get older.

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.

3.3.1 Implications

For researchers: We found that average commit size slightly decreases over time. This could be explained by different development practices applied during different software development stages (development, testing, support, etc). Our recommendation for researchers is to further investigate how different software development stages shape development practices.

DVCS repositories have more ITL in their commits. This fact suggests that DVCS repositories are better candidates for research projects studying links between commits and issue tracking systems.

For tool builders: As long as changes tend to become smaller as project matures, it is easier and faster to rebuild part of applications that were changed instead of rebuilding whole application. Thus, we encourage tool builders to provide more support for incremental builds (compiling and rebuilding only parts of the application that were changed) in IDEs and build management tools.

Changes are more granular in DVCSs and usually have only one issue reference. Therefore, for DVCS we expect implementation of such feature as cherry-picking by issues (instead of cherry-picking by commits) to be more reliable than for CVCS.

One of the reasons why developers do not put issue numbers into commits could be the extra work it involves. It usually means switching from the commit interface and looking into the ITS to see what the issue number is. This is a distraction developers might want to avoid. Better integration between Version Control Systems and ITS can mitigate this problem. It will allow developers to find the reference they are looking for more easily. Also, tools suggesting the issue to reference could be useful.

For team management: As long as commit size tends to become smaller as projects get older, it is reasonable to assume that at later stages in the project developers tend to spend more time thinking about the source code instead of adding more lines of code. Therefore, developers productivity should be measured not only by the amount of code they produce, but also by the complexity and importance of their changes.

If the company policy requires mandatory ITL in a commit message, switch to DVCS might help developers with assigning issue tracking labels more easily and consistently.

4. 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?

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. [33] 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.

Also, there is a chance that one developer might use different name aliases when committing in the same repository. This could affect metrics that rely on team size for repositories analysis. Even though we have encountered few cases of aliases usage while analyzing repository data, we also noticed that it is a very rare and exceptional practice.

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 [18] 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.

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 the 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, 80% of the projects were developed in Java, and the remaining 10% used C/C++ and Javascript. While we have no reasons to believe that programming language affects the culture of committing changes, in the future we pan to diversify our corpus.

The sources for our repositories are GitHub and SourceForge. This means that we only looked at projects that used Git or SVN. Thus, we did not study several 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.

Reliability: Can others replicate our results? The list of repositories we used for our analysis is available online [11]. Also, the infrastructure we used for the analysis is available open source as a GitHub repository [5].

5. RELATED WORK

To the best of our knowledge, our paper is the first study to *compare* the impact of CVCS and DVCS on the practice of committing changes.

Several researchers [12, 13, 19, 22, 24, 25, 29, 32, 32] studied the practice of commits but only in the CVCS paradigm. Purushothaman et al. [32] and German et al. [19] and Hindle et al. [24] studied the properties of typical small commits or typical large commits. Hattori et al. [22] study the size of commits with the purpose of classifying changes. Arafat et al. [13] studied the distribution of commit size. Hofmann et al. [25] predict commit size based on commit history. Herzig et al. [23] propose an algorithm for untangling changes in CVCS. However, ours is the first study to compare the commit size in CVCS and DVCS.

Related with our study about the impact and the presence of issue tracking systems (ITS), several researchers [14, 16, 21, 30, 35] studied ITS. Tian et al. [35] and Bird et al. [16] aim to link source code with ITS. Meneely et al. [30] makes suggestions on improving issue tracking labeling in commit messages. Bachmann et al. [14] mine bug tracking databases and with the purpose of linking ITS with the software development process. Hassan et al. [21] provide an overview of repository and ITS mining practices. However, none of these studies compared CVCS and DVCS based on ITS practices.

Recently, researchers started mining DVCS repositories for collaboration between developers [34], processes [17, 20] for mining DVCS repositories

6. CONCLUSIONS

DVCS have taken the software development world by storm. On the other hand, the traditional, centralized VCS are still used. We need to understand if this shift in paradigm is evolutionary or revolutionary.

In this study we used two data sources: a survey of 820 participants and a corpus of 132 repositories. Our comparison of CVCS and DVCS reveals that they have observable effects on developers, teams, and processes. The most surprising findings are that (i) the size of commits in DVCS was 32% smaller than in CVCS, (ii) developers split commits more often in DVCS, and (iii) DVCS commits are more likely to have references to issue tracking labels.

We hope that our work inspires future research into the impact that centralized and distributed VCS tools have on software development.

7. ACKNOWLEDGEMENTS

We would like to 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 and Paul McKenney for feedback on earlier drafts of this paper.

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

This research is partly funded through a grant from the National Science Foundation, CCF-1213091.

8. REFERENCES

[1] Cvs. `http://cvs.nongnu.org/`. Accessed September 13, 2013.

[2] Git. `http://git-scm.com/`. Accessed September 13, 2013.

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

[4] Github. `http://www.github.com/`. Accessed September 13, 2013.

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

[6] Mercurial. Accessed September 13, 2013.

[7] Sourceforge. `http://www.sourceforge.net/`. Accessed September 13, 2013.

[8] 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.

[9] Svn. `http://subversion.tigris.org/`. Accessed September 13, 2013.

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

[11] Vcs usage study companion. `http://tiny.cc/VCStudy`.

[12] A. Alali, H. Kagdi, and J. 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.

[13] O. Arafat and D. 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.

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

[15] C. Bird, A. Bachmann, E. Aune, J. Duffy, A. Bernstein, V. Filkov, and P. 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.

[16] C. Bird, A. Bachmann, F. Rahman, and A. 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.

[17] C. Bird, P. C. Rigby, E. T. Barr, D. J. Hamilton, D. M. German, and P. 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.

[18] D. S. Cruzes and T. 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.

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

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

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

[22] L. P. Hattori and M. 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.

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

[24] A. Hindle, D. M. German, and R. 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.

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

[26] J. Janák. Issue tracking systems. *Brno, spring*, 2009.

[27] M. Lanza and R. 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.

[28] R. 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.

[29] M. Marzban, Z. Khoshmanesh, and A. Sami. Cohesion between size of commit and type of commit. In *Computer Science and Convergence*, pages 231–239. Springer, 2012.

[30] A. Meneely, M. Corcoran, and L. 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.

[31] S. Phillips, J. Sillito, and R. Walker. Branching and merging: an investigation into current version control practices. In *Proceedings of the 4th International Workshop on Cooperative and Human Aspects of Software Engineering*, CHASE '11, pages 9–15, New York, NY, USA, 2011. ACM.

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

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

[34] P. C. Rigby, E. T. Barr, C. Bird, P. Devanbu, and D. M. German. What effect does distributed version control have on oss project organization?

[35] Y. Tian, J. Lawall, and D. Lo. Identifying linux bug fixing patches. In *Software Engineering (ICSE), 2012 34th International Conference on*, pages 386–396. IEEE, 2012.

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