

CRISTIANO MACHADO CESÁRIO

AWARENESS OVER DISTRIBUTED VERSION CONTROL SYSTEMS

Thesis presented to the Computing Graduate program of the Universidade Federal Fluminense in partial fulfillment of the requirements for the degree of Master of Science. Topic Area: Software Engineering.

Advisor: Prof. D.Sc. Leonardo Gresta Paulino Murta

Niterói

2015

Ficha Catalográfica – Esta página deve ser removida na versão a ser entregue para a banca, mas deve ser reinserida na versão final, com a ficha catalográfica fornecida pela biblioteca. Informações sobre este processo devem ser obtidas na secretaria da pós-graduação.

CRISTIANO MACHADO CESÁRIO

AWARENESS OVER DISTRIBUTED VERSION CONTROL SYSTEMS

Thesis presented to the Computing Graduate program of the Universidade Federal Fluminense in partial fulfillment of the requirements for the degree of Master of Science. Topic Area: Software Engineering.

Approved on April 2015.

APPROVED BY

Prof. D.Sc. Leonardo Gresta Paulino Murta – Advisor
IC-UFF

Prof. D.Sc. Daniel Cardoso Moraes de Oliveira
IC-UFF

Prof. D.Sc. Flávia Coimbra Delicato
DCC-UFRJ

Niterói
2015

For Agostinho, my father, and Ondina, my mother.

ACKNOWLEDGMENTS

First of all, I am grateful to God, who infused in me the intelligence to be able to complete this thesis.

I would like to thank my parents, Ondina and Agostinho, who supported my education, especially during my bachelor degree, both financially and with advises.

I wish to express my deepest gratitude to my wife, Raquel, who encouraged me to apply for this Master of Science degree, and who had so many lonely moments by my side, while I was involved with my research. I love you, and will always be grateful for understanding the adversity we went through.

I place on record my sincere thanks to my advisor, Leonardo. His help, counsels, patience, and tireless dedication were decisive to come to this thesis.

I also thank my fellow postgraduate students in the computer science department, especially to Ivison who was always available to listen to my concerns and to share his life experience with everybody.

I also place on record my thanks to the members of the committee, Flávia and Daniel, for spending their time to evaluate this thesis.

RESUMO

O desenvolvimento de *software* utilizando sistemas de controle de versão distribuídos tem se tornado recentemente cada vez mais frequente. Tais sistemas trazem mais flexibilidade, mas também trazem uma maior complexidade para administrar e monitorar os múltiplos repositórios existentes, além de induzir à proliferação de vários ramos.

Neste trabalho, propomos o DyeVC, uma abordagem para auxiliar desenvolvedores e administradores de repositórios a identificar dependências entre os clones de repositórios distribuídos, como forma de ajudar a entender o que acontece ao redor do clone de alguém e descobrir as relações entre os clones existentes. Um protótipo foi desenvolvido, para aplicar a abordagem proposta. Dois experimentos e um estudo observacional foram realizados com o intuito de avaliar a abordagem DyeVC.

Palavras-chave: Gerência de configuração, Percepção de espaços de trabalho, Sistema de Controle de versão distribuído, Evolução de repositórios

ABSTRACT

Software development using distributed version control systems has become more frequent recently. Such systems bring more flexibility, but also bring greater complexity to administer and monitor the multiple existing repositories, and induce the proliferation of several branches.

In this work, we propose DyeVC, a tool to assist developers and repository administrators in identifying dependencies amongst clones of distributed repositories in order to help to understand what is going on around one's clone and depict the relations between the existing clones. A prototype was developed, applying the proposed approach. Two experiments and an observational study were conducted to evaluate DyeVC.

Keywords: Configuration management, Workspace awareness, Distributed version control system, Repository evolution

LIST OF FIGURES

Figure 1 – A development scenario involving some developers.....	17
Figure 2 – CVCS (a) versus DVCS (b) (MURTA, 2012)	22
Figure 3 – Commits in a repository	24
Figure 4 – Branch “issue55” is created and development occurs in parallel	24
Figure 5 – Branching through cloning.....	25
Figure 6 – Repositories after pushing changes from clone A.....	26
Figure 7 – Repositories after merging work from clone B.....	26
Figure 8 – Repositories after pushing changes from clone B.....	26
Figure 9 – Repositories after merging work from clone C.....	26
Figure 10 – Branch after fetching updates from the original repository	27
Figure 11 – Result of pushing changes from a tracked branch.	27
Figure 12 – Anonymous branch	28
Figure 13 – Commit notification approaches	29
Figure 14 – Eclipse window with Palantir plugin (SARMA <i>et al.</i> , 2012)	30
Figure 15 – CollabVS snippets (DEWAN; HEGDE, 2007).....	31
Figure 16 – FASTDash Visualization Runtime (BIEHL <i>et al.</i> , 2007)	32
Figure 17 – Lighthouse plugin on Eclipse (DA SILVA <i>et al.</i> , 2006).....	33
Figure 18 – WeCode continuous merging (GUIMARÃES; SILVA, 2012).....	34
Figure 19 – Crystal snapshot (BRUN <i>et al.</i> , 2011).....	34
Figure 20 – Gevol: focus on program structures (COLLBERG <i>et al.</i> , 2003).....	35
Figure 21 – The Evolution Matrix: focus on classes (LANZA, 2001).....	35
Figure 22 – CVSScan: focus on lines (VOINEA <i>et al.</i> , 2005).....	36
Figure 23 – LifeSource: focus on authors (GILBERT; KARAHALIOS, 2006).....	37
Figure 24 – Polvo: focus on branches (SANTOS; MURTA, 2012).....	37
Figure 25 – VisGi: focus on branches (ELSEN, 2013)	38
Figure 26 – Visugit: focus on branches (HOZUMI, 2010)	38
Figure 27 – GitHub’s Network Graph: focus on branches (PRESTON-WERNER, 2008)	39
Figure 28 – gitk client.....	40
Figure 29 – TortoiseGit client	40
Figure 30 – How DyeVC gathers information	44
Figure 31 – DyeVC discovering the topology.....	45
Figure 32 – Remote repository configuration in Git’s <i>config</i> file	45

Figure 33 – Model used to store topology data	46
Figure 34 – DyeVC showing notifications in the notification area	47
Figure 35 – Topology view of DyeVC project, at a given moment	48
Figure 36 – DyeVC main screen	49
Figure 37 – Developers led by Wolverine	50
Figure 38 – Commit history for a given project	51
Figure 39 – Collapsed commit history	53
Figure 40 – Topology view showing first monitored repository (Sep 24 2010)	63
Figure 41 – aakoch's commit history showing commits pending to be pushed.....	63
Figure 42 – Topology view showing the three monitored repositories (Sep 27 2010)	63
Figure 43 – Adam's tracked branches	65
Figure 44 – Jeresig's collapsed commit history	65
Figure 45 – Aakoch's commit history	65
Figure 46 – Jeresig's tracked branches.....	66

LIST OF TABLES

Table 1 – Possible states of a repository.....	50
Table 2 – Status of a local repository regarding a remote one, based on the existence of non-replicated commits.....	50
Table 3 – Existing commits in each repository	51
Table 4 – Status of each repository based on known remote repositories.....	51
Table 5 – Comparing DyeVC features with related work	58
Table 6 – Summary of the Characterization Form	68
Table 7 – Time spent to answer each question in the study	69
Table 8 – Expected answers to questions proposed in both phases.....	70
Table 9 – Monitored projects and repository metrics taken during evaluation	73
Table 10 – Time spent to perform foreground operations.....	74
Table 11 – Time taken to perform background operations	74
Table 12 – Pearson coefficient between time spent and repository metrics for measured operations	75

LIST OF ACRONYMS AND ABBREVIATIONS

API	– Application Programming Interface
CI	– Configuration Item
CM	– Configuration Management
CVCS	– Centralized Version Control System
DAG	– Directed Acyclic Graph
DVCS	– Distributed Version Control System
JSON	– JavaScript Object Notation
RCS	– Revision Control System
RESTful	– Representational State Transfer
SCCS	– Source Code Control System
VCS	– Version Control System

TABLE OF CONTENTS

Chapter 1 – Introduction.....	15
1.1 Motivation.....	15
1.2 Goals	18
1.3 Organization.....	19
Chapter 2 – Awareness over Distributed Version Control Systems.....	21
2.1 Introduction.....	21
2.2 Distributed Version Control Systems	22
2.3 Branching in DVCS	23
2.3.1 Cloning a repository	24
2.3.2 Push and pull changes	25
2.3.3 Branch tracking	27
2.3.4 Anonymous branches	27
2.4 Related Work	28
2.4.1 Commit notification	29
2.4.2 Awareness of concurrent changes	30
2.4.3 Repository visualization.....	35
2.4.4 DVCS clients.....	39
2.5 Final considerations	41
Chapter 3 – Approach.....	42
3.1 Introduction.....	42
3.2 Information gathering	43
3.3 Information visualization.....	47
3.3.1 Level 1: Notifications.....	47
3.3.2 Level 2: Topology	48
3.3.3 Level 3: Tracked branches	49
3.3.4 Level 4: Commits.....	51

3.4 How information is gathered	53
3.5 Implementation details.....	56
3.6 Final considerations	57
Chapter 4 – Evaluation	61
4.1 Introduction.....	61
4.2 Analyzing <i>JQuery</i> project with DyeVC.....	61
4.3 Observational study	66
4.3.1 Description	67
4.3.2 Procedure.....	68
4.3.3 Results	68
4.3.4 Subjects evaluation.....	71
4.4 Performance evaluation	73
4.5 Threats to validity	76
4.6 Final considerations	77
Chapter 5 – Conclusion	78
5.1 Contributions	78
5.2 Limitations	78
5.3 Future work.....	80
Bibliography	82
Appendix A – Commit History Visualization	87
Appendix B – DyeVC Usage	89
B.1 Introduction	89
B.2 Running DyeVC.....	89
B.2.1 Main window	89
B.2.2 Monitored repositories.....	90
B.2.3 Visualizations	91
B.3 Typical usages	92

B.4 Further configurations	93
B.4.1 Refresh rate.....	93
B.4.2 Connecting to a different database	94
B.4.3 Clearing the cache	94
Appendix C – Informed Consent Form	95
Appendix D – Characterization Form	97
Appendix E – Activities – Phase 1	99
Appendix F – Activities – Phase 2	101
Appendix G – Exit Survey	107

CHAPTER 1 – INTRODUCTION

1.1 MOTIVATION

Version Control Systems (VCS) date back to the 70s, when Source Code Control System (SCCS) emerged (ROCHKIND, 1975). Their primary purpose is to keep software development under control (ESTUBLIER, 2000). Along these 40 years, VCSs evolved from a centralized repository with local access, as in SCCS and Revision Control System (RCS) (TICHY, 1985), to a client-server approach, as in CVS (CEDERQVIST, 2005) and Subversion (COLLINS-SUSSMAN *et al.*, 2011). More recently, distributed VCSs (DVCS) arose, allowing clones of the entire repository in different locations, as in Git (CHACON, 2009) and Mercurial (O’SULLIVAN, 2009b). According to a survey conducted among the Eclipse community (ECLIPSE FOUNDATION, 2013), Git and Github combined usage increased from 6.8% to 36.3% between 2010 and 2013 (a growth greater than 600%). During this same period, Subversion and CVS combined usage decreased from 71% in 2010 to 42.3% in 2013. This clearly shows the momentum and a strong tendency in the adoption of DVCSs among the open source community.

Besides these changes from local to client-server and then to a distributed architecture, the concurrency control policy adopted by VCSs changed from lock-based (pessimistic) to branch-based (optimistic). According to Walrad and Strom (2002), creating branches in VCSs is essential to software development because it enables concurrent development, allowing the maintenance of different versions of a system in parallel, the customization to different platforms and to different customers, among other features that are expected by current software development teams. DVCS include better support for working with branches (O’SULLIVAN, 2009a), turning the branch creation into a recurring pattern, no matter if this creation is explicitly done by executing a “*branch*” command or implicitly when a repository is cloned. All these branches, whether explicit or not, will eventually be reintegrated by means of merge operations, reflecting to the main development line the changes made.

DVCS usage typically follows a *push/pull* model (more details in Section 2.2). Assuming an existing repository named *rep*, one can create a repository clone of *rep* (*rep'*, for example). This clone is in fact a mirror of *rep*, containing all commits that exist in *rep* by the time when *rep'* was created. Commits can be concurrently created in *rep* and in *rep'*, and to maintain both repositories updated, one working with *rep'* should periodically bring remote

commits from *rep* by means of a *pull* command and send local commits to *rep* by means of a *push* command.

However, distributed software development, especially from the geographical perspective (GUMM, 2006), brings a set of risk factors, and Configuration Management (CM) is affected by them (BATTIN *et al.*, 2001). The increasing growth of development teams, and their distribution along distant locations – even different continents – together with the proliferation of branches, introduce additional complexity for perceiving actions performed in parallel by different developers. According to Perry *et al.* (1998), concurrent development increases the number of defects in software. Besides, Silva *et al.* (2006) say that branches are frequently used for promoting isolation amongst developers. This postpones the perception of conflicts that result from changes made by co-workers, as these conflicts are noticed only after a pull or a push. Moreover, Brun *et al.* (2011) show that, even using modern DVCSs, conflicts during *merges* are frequent, persistent, and appear not only as overlapping textual edits (i.e., physical conflicts) but also as subsequent build (i.e., syntactic conflicts) and test failures (i.e., semantic conflicts). Even when two developers are working in different features, changing different artifacts, this could lead to conflicts. For example, if a developer changes the behavior of a method in a superclass, all subclasses of this class that rely on that method will be potentially affected by this behavior change.

By enabling repository clones, DVCS expand the branching possibilities discussed by Appleton *et al.* (1998), allowing several clones to coexist with fragments of the project history. This may lead to complex topologies where changes can be sent to or received from any clone. This scenario generates traffic similar to that of peer-to-peer applications. In practice, projects impose some restrictions over this topology freedom. However, it can be still much more complex than the traditional client-server topology found in Centralized Version Control Systems (CVCS).

To illustrate this situation, Figure 1 shows a scenario with some developers, each one owning a clone of the repository originally created at Xavier Institute. Xavier Institute acts like a central repository, where code developed by all teams is integrated, tested, and released to production. There is a team working at Xavier Institute, led by Professor Xavier. Outside the Institute, Wolverine leads a remote team located in a different site, which is constantly synchronized with the Institute. Solid lines in Figure 1 indicate data being pushed, whereas dashed lines indicate data being pulled. Thus, for example, Rogue can both pull updates from Gambit and push updates to him, and Beast can pull updates from Rogue, but cannot push updates to her. It is not common to have a scenario where pushes are performed from a

developer to another (such as the ones between Beast and Gambit, Rogue and Gambit, Nightcrawler and Rogue). Generally, what happens is that a developer pulls from another (for example, among Cyclops, Jean Gray and Mystique). This is to avoid that a developer inadvertently creates commits inside another's clone. Although infrequent, this scenario helps in understanding the need to have awareness about who are the peers in a project and what are their interdependencies.

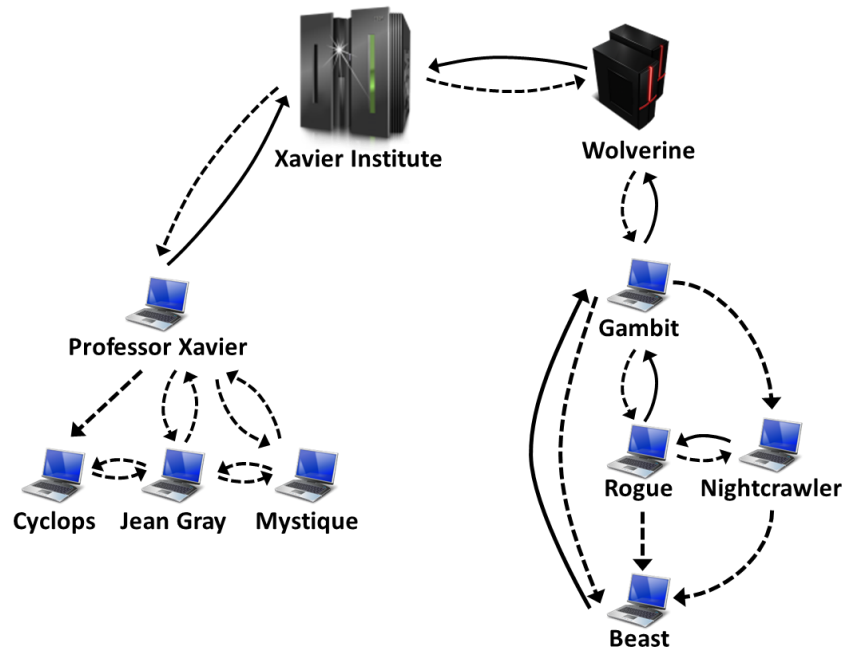


Figure 1 – A development scenario involving some developers

Each one of the developers has a copy of the repository. Luckily, this scenario has a CM Plan in action, otherwise any one would be able to send and receive updates to or from each other. In addition to the existence of a CM Plan, there are a number of workflows established to work with DVCSs, such as the *Centralized Workflow*, the *Integration-Manager Workflow*, and the *Dictator and Lieutenants Workflow* (CHACON, 2009), that could help in organizing the flow of commits. Otherwise, this would lead to a total of $n \cdot (n - 1)$ different possibilities of communication (where n is the number of developers in the topology). In practice, this limit is usually not reached: while interaction amongst some developers is frequent, it may happen that others have no idea about the existence of some coworkers. It occurs with Mystique and Nightcrawler, for example, where there is no direct communication.

As an example, from a developer's point of view, like Beast, how can he know at a given moment, if there are commits in Rogue, in Gambit, or in Nightcrawler clones that were not pulled yet? Alternatively, would be the case that there are local commits pending to be pushed to Gambit? Beast could certainly periodically pull changes from his peers, checking if

there were updates available, but this would be a manual procedure, error prone. DVCSs generally require that the user perform a pull before a push, whenever there are commits not pulled yet, but this will postpone the awareness of changes to a moment where local changes were already finished and are ready to be pushed. It would be more practical if Beast could have a continuously up-to-date knowledge of his peers, warning him about any local or remote updates that had not been synchronized yet.

On the other hand, from an administrator’s point of view, how can they know the existing clones of a project and how they relate among each other? How can they know if there are pending commits to be sent from a staging clone to a production one? This kind of perception regarding others work is known as “awareness”, which is defined by Dourish and Bellotti (1992) as “an understanding of the activities of others to provide a context for one’s own activities”.

There are many approaches that aim at providing awareness of concurrent work using VCSs, such as *Palantir* (SARMA *et al.*, 2012), *FASTDash* (BIEHL *et al.*, 2007), *Lighthouse* (DA SILVA *et al.*, 2006), and *CollabVS* (DEWAN; HEGDE, 2007). The majority of these approaches, though, are focused only in CVCSs, which are much less prone to branches if compared to DVCSs. The only approach found that focus on DVCSs is *Crystal* (BRUN *et al.*, 2011), which continuously merge work from registered peers into a local clone, reporting to the user conflicts eventually found. However, Crystal does not deal with different branches automatically, demanding that the user creates a different project for each branch that they want to monitor. Moreover, Crystal does not offer a way to discover dependencies between clones (i.e., peers).

1.2 GOALS

In this work, we propose a novel monitoring and visualization infrastructure for DVCS, named DyeVC¹, which gathers information about different repository clones, consolidates this information, and presents them visually to the user, allowing one to perceive how their clone evolved over time and how this evolution compares to the evolution of other clones.

Thus, this work proposes a platform that enables repository administrators to monitor and visualize which the existing clones of a project are and how they interact with each other. The information provided by our approach is important for a number of reasons, such as:

¹ Dye is commonly used in cells to observe the cell division process. As an analogy, DyeVC (Dye over Version Control) allows developers to observe how a Version Control repository evolved over time.

- It allows the configuration manager to verify if communication is taking place accordingly, based on what was defined in the CM Plan;
- It helps administrators and developers in knowing who the participating peers in a project are and how they depend upon each other;
- It increases the developer knowledge of what is going on around their repository clone and the clones of their teammates, despite the branch where changes are being done.

Our intention is that, for a given project, our approach be capable of answering the following questions:

- Q1: Which clones were created from a repository?
- Q2: What are the dependencies between different clones?
- Q3: Which changes are under work in parallel (in different clones or different branches) and which of them are available to be incorporated into my work?

Besides these questions, the following question is posed, which is related to the non-functional aspects of our approach:

- Q4: Is it computationally feasible to gather this information from all known repository clones, keeping them available to be used when needed?

1.3 ORGANIZATION

Besides this introduction, this work is organized in four other chapters. Chapter 2 presents some introductory topics regarding DVCS. It contrasts DVCS usage against CVCS. It also explains the concept of *branches* and how they are used in DVCS. Lastly, it presents the related work, which include commit visualization approaches, approaches that provide awareness of concurrent changes, approaches that focus on repository visualization and commercial / open source DVCS clients.

Chapter 3 presents DyeVC. This chapter describes how DVCS information is gathered and structured. Then it outlines the existing visualizations in a hierarchical way, discussing the level of detail included in each one, over the example introduced in Section 1.1. It also discusses the algorithm used in information gathering, which is in the heart of the process that discovers related peers, dependencies, and commits found in each peer. Furthermore, it presents the

technologies used in the implementation. Finally, it shows a typical usage scenario of DyeVC, describing its prototype and the first steps needed to use it.

Chapter 4 describes the evaluation performed on the usage of DyeVC to provide awareness over an open source project that uses DVCS. Next, the scalability of the approach is evaluated, presenting the factors that may affect the capability of using DyeVC. Lastly, it presents some threats to the validity of the performed evaluation.

Finally, Chapter 5 concludes this work, presenting contributions, limitations, and future work.

CHAPTER 2 – AWARENESS OVER DISTRIBUTED VERSION CONTROL SYSTEMS

2.1 INTRODUCTION

Configuration management (CM) is part of Software Engineering and was born in the 70's (ESTUBLIER, 2000). According to Murta (2006), under the development perspective, CM is divided into three main systems: Change Management, Version Control, and Release Management. **Change Management** is in charge of systematically controlling the configuration, storing and reporting the information produced along the change requests. **Version Control** deals with the identification and evolution of configuration items (CI). Finally, **Release Management** automates the process of building executable files from the source code and releasing them into production.

Version Control Systems (VCSs) date back to the 70's, when the first VCS, called SCCS, emerged (ROCHKIND, 1975). VCSs were the first CM systems to emerge and, since then, they have evolved substantially, from the local access needed by SCCS, to a client-server architecture, and more recently to a distributed architecture. Besides their original scope, they have also been used as a data source to mine data related to software development, because they have the knowledge of all CIs and how they evolved over time (what was changed, why changes occurred, when changes happened, and who performed the changes). Moreover, as VCSs are the single point where every CI resides, they might be used to not only know what happened in the past, but also provide awareness of what is happening in the present and to help predicting what might happen in the future.

Awareness is defined by Dourish and Bellotti (1992) as “an understanding of the activities of others to provide a context for one's own activities”. The information needed to provide awareness depends on what people need to be aware of. In the context of this work, which focus on the awareness of changes occurring in DVCS, a portion of the information needed is available in the repository itself, such as which files have changed, what changes were applied, who applied the changes, when the changes occurred, and why the changes were applied. Another portion of the information needed, though, has to be discovered and consolidated in order to be used, such as who are the existing peers in the project, how they communicate with each other and which changes can be found in each one of them.

This chapter presents some basic concepts related to DVCS and some approaches related to providing awareness combined with VCS. Section 2.2 discusses central concepts

regarding how DVCS work. Section 2.3 covers several branching situations that occur when working with DVCS. Section 2.4 presents related work. Finally, Section 2.5 presents the final considerations of this chapter.

2.2 DISTRIBUTED VERSION CONTROL SYSTEMS

CVCS relies on a centralized repository, stored on a server (Figure 2.a). When someone wants to work on any CI, a *checkout* is performed, copying a specific version of the artifacts from the repository to a workspace where changes can be applied. Later, after applying the changes, a *commit* (also known as *check-in*) is performed, sending all the changes back to the repository. The group of changes a commit introduces is referred as a *changeset*. Updates made by other developers can be brought to the workspace at any time, by performing an *update*. Updates are automatically merged into workspace, or, in the case of physical conflicts (same part of the artifact changed locally and remotely), the developer might have to handle the merge manually.

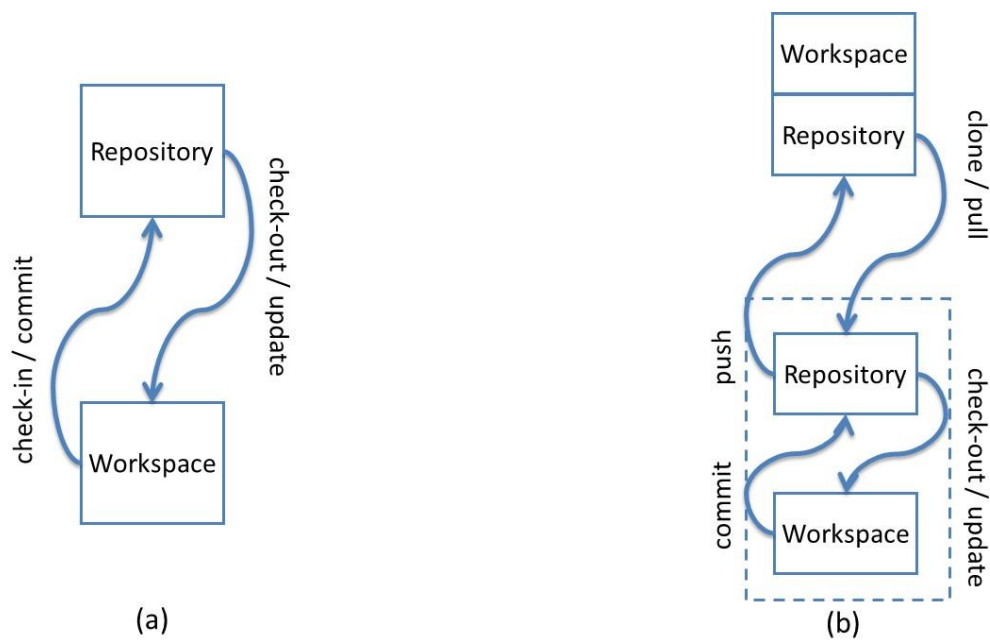


Figure 2 – CVCS (a) versus DVCS (b) (MURTA, 2012)

DVCS, on the other hand, does not rely on a centralized repository. It uses an architecture where the entire repository is distributed and it exists in every machine where someone wants to work with it (leading to the existence of several clones of a repository). The changes continue to take place on a workspace, but there is a local copy of the repository attached to the workspace (Figure 2.b). The main operations (*checkout*, *commit*, and *update*) continue to exist, but are performed locally, allowing offline work, committing whenever necessary. However, another set of commands arises, which allows sending and receiving

changes between different clones. Initially, a *clone* command is performed, copying the repository from a specified location. The copy can be done from a repository located on a partner or on a server. We will refer to the original repository as *remote repository*. There is no concept of a *central* repository, but some repository may act as a central repository by having, for example, a strict policy regarding who might send changes to it. Repositories might either have an associated workspace or not. A repository located on a server, where local changes are not expected to occur, does not need to have a workspace associated, and is referred as a *bare* repository. A *bare* repository is just like a regular one, except that it does not have a workspace associated with it. Due to the lack of a workspace associated, commands that depend upon a workspace (*commit*, *update*, and *merge*) cannot be performed on a *bare* repository. Changes can be sent from a local repository to a remote repository by invoking a *push* command. Changes can be received from remote repositories and applied immediately to the workspace, leading to merges / conflicts, or they can stay at the local repository to be applied later. A *pull* command brings the changes and applies them to the workspace. A *pull* can be broken into two subcommands²: a *fetch* command, that only transfers the changes to the local repository, without applying them to the workspace, and a *merge* command, which applies the changes to the workspace.

2.3 BRANCHING IN DVCS

The commits performed in a repository are based on previous commits, thus forming a directed acyclic graph (DAG), where newer commits point to older commits. Figure 3 represents a repository with three commits. Commit “3” is pointed by a label named *master*. As new commits arise, the *master* pointer moves accordingly. This way, if one wants to refer to the last commit in this repository, they do not need to know how many commits are there. All they have to do is to ask for the commit pointed by the *master* label.

This linear development is not frequent in real repositories, for a number of reasons, such as supporting multiple releases of the same software, trying new technologies that might be included in future versions or fixing multiple bugs in parallel³. In such cases, it is common

² Here we focus on the possibility of breaking a *pull* into two different commands, but keep in mind that different tools can have different commands to do the same operation. An example is the *pull* command. Whereas in *Git* it brings the changes and applies them to the workspace, in *Mercurial* it only brings the changes and we have to perform an *merge* command to apply them. Later versions of *Mercurial* are being distributed with a *fetch* extension, which brings the changes and applies them. This way, the *pull* command in *Git* is equivalent to the *fetch* command in *Mercurial*, and vice-versa. Whenever we mention any command herein, we will be expecting the behavior provided by *Git*.

³ APPLETON *et al.* (1998) present a compilation of branching patterns to address different needs.

to use *branches*, which are, according to Leon (2004), “a deviation from the main development line for an item”.

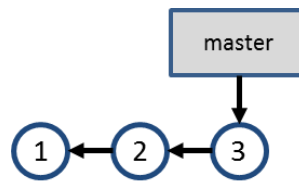


Figure 3 – Commits in a repository

Suppose that a bug is found in a software product during tests. The test team creates issue 55, asking the development team to fix the bug. The development team verifies that the software version being tested corresponds to commit #3, but the bug will take a considerable time to be fixed and the development cannot stop waiting for the bug fix. A branch named “issue55” is then created, pointing to commit #3 (Figure 4.a). At this moment, development can occur in parallel, with commits #5 and #7 being made to branch “issue55” and commits #4 and #6 being made to branch “master” (Figure 4.b). Finally, the team decides to merge the fix to issue #55 into the “master” branch, generating commit #8 (Figure 4.c).

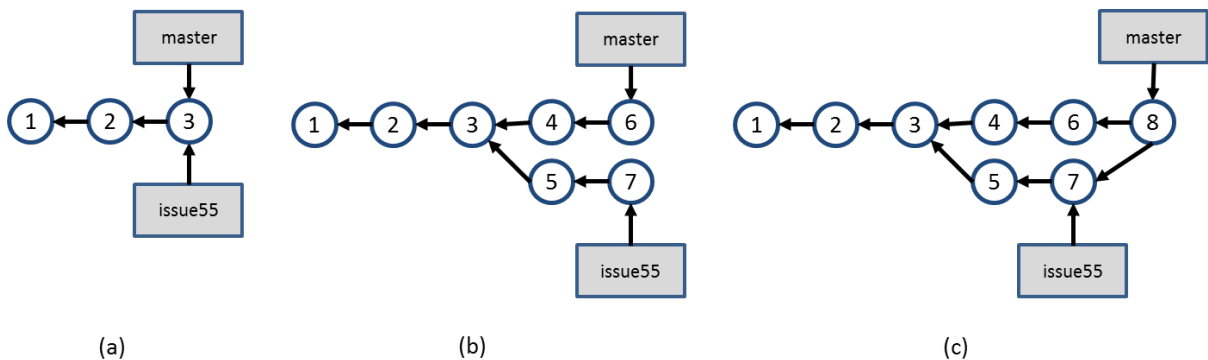


Figure 4 – Branch “issue55” is created and development occurs in parallel

This is a scenario that shows the usage of *explicit branches*, i.e., a scenario where one explicitly creates a branch to perform some work. However, when using DVCS, there are other situations where branches arise implicitly. In these situations, one does not explicitly asks to create a new branch, but the branch concept is used anyway. We refer to branches that are not created explicitly as *implicit branches*. The next Sections will show some situations where this occurs.

2.3.1 CLONING A REPOSITORY

Suppose a repository that has a history like the one shown in Figure 3. At this point, clones A, B, and C are created by different developers, as shown in Figure 5, where the first three commits are the same in all of them, but other commits are created, each one existing in

only one of the clones. The colors in the commits are used to identify the clone where that commit was originally done. The numbers identifying each commit are used for the sake of this example, but commits are not necessarily identified by a sequential number. For example, in Git, commits are identified by a hash code of 40 hexadecimal digits.

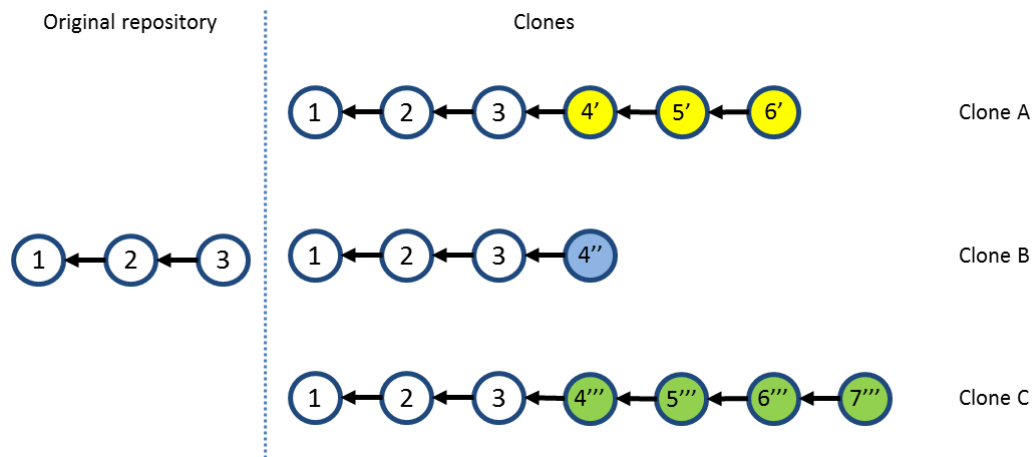


Figure 5 – Branching through cloning

As discussed in Section 2.2, each clone is a complete repository copy, independent from the other clones, enabling parallel development. Throughout the text, we will use the terms *clone*, *repository clone*, or simply *repository*, to refer to any known clone of an original repository, except when we refer to a specific clone (for example, a *remote repository* or a *central repository*). We can say that each clone is a *fork* of the original *repository*, although we say that a repository is a *clone* if it is located at a developer's machine, and it is a *fork* when it is located at a server. Furthermore, it is important to distinguish the *clone* command (verb) from a clone (subject) created by means of a *clone* command.

2.3.2 PUSH AND PULL CHANGES

Having a number of existing clones, the first one to push changes to the original repository is able to do it with no extra work. Looking at Figure 5, let us assume that clone A is the first one to push its changes to the original repository. This results in a scenario like the one shown in Figure 6.

Next, clone B tries to push its changes to the original repository, but this is not allowed, because it could result in inconsistencies in the VCS, once commits from clone A were never tested together with commits from clone B. To allow clone B to push its changes, it is necessary to first pull commits 4', 5' and 6' from the original repository, merging them into the clone B workspace, resulting in the scenario shown in Figure 7. Notice that, in this case, there is an additional commit, denoted by 5''.

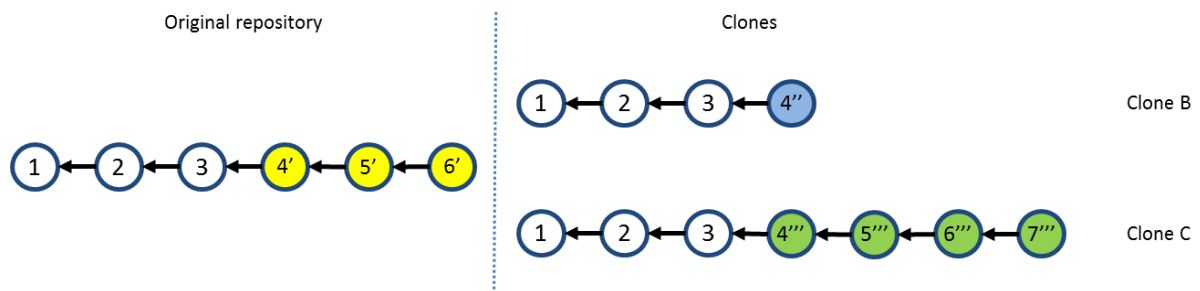


Figure 6 – Repositories after pushing changes from clone A

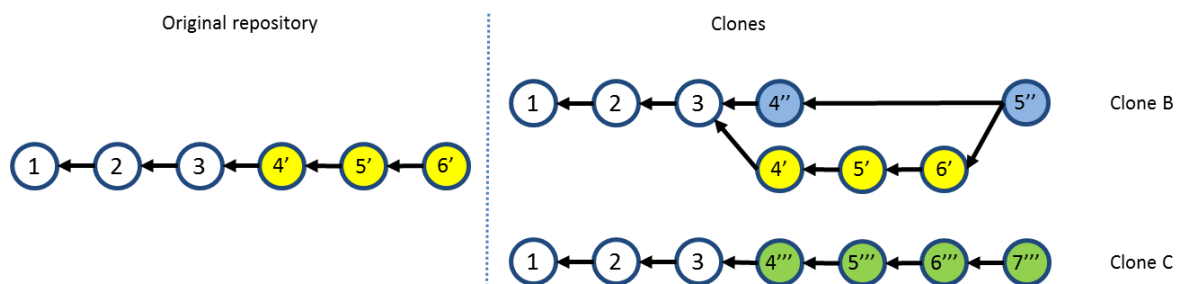


Figure 7 – Repositories after merging work from clone B.

Clone B then pushes its changes, resulting in the scenario shown in Figure 8. Finally, clone C has to follow the same procedure, i.e., pull the changes previously pushed by clone A and clone B, and merge them into its workspace, resulting in the scenario shown in Figure 9. As a final step, clone C would push its changes, and the original repository would then have the same set of commits as clone C.

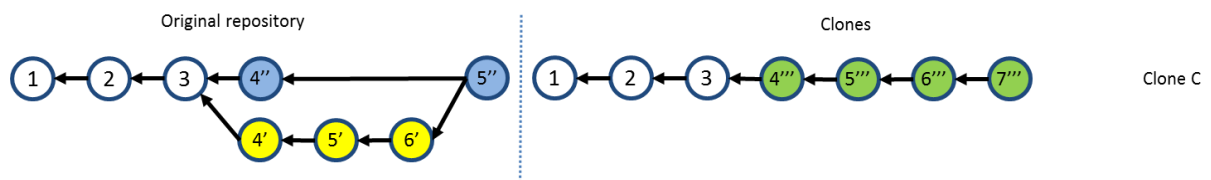


Figure 8 – Repositories after pushing changes from clone B

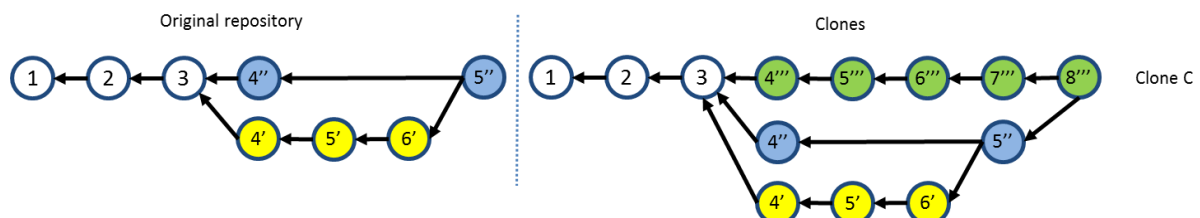


Figure 9 – Repositories after merging work from clone C

Besides pulling changes, which merges the changes them into the workspace, one could choose to bring the changes without immediately merging them. This is accomplished by only fetching the updates. If clone B had chosen only to fetch updates from the original repository,

this would result in a branch, as shown in Figure 10. Here, branch *master* is the local branch in clone B, and branch *origin/master* is the master branch from the original repository.

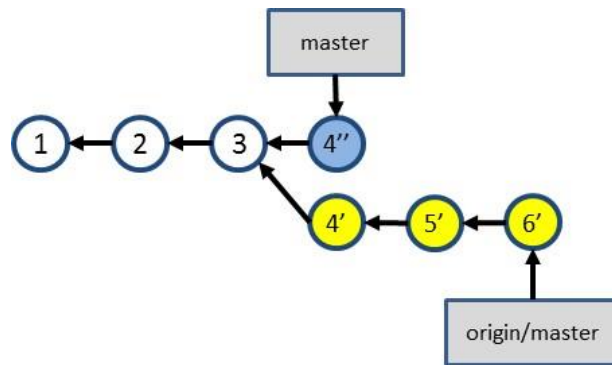


Figure 10 – Branch after fetching updates from the original repository

2.3.3 BRANCH TRACKING

Another distinction between branches is that of *tracked* and *non-tracked* branches. A non-tracked branch is a local branch that is not shared with peers. A non-tracked branch exists only in the local repository and it is not pushed to remote repositories. On the other hand, a tracked branch is a branch that one chooses to share with peers, by associating a local branch with a branch in the remote repository. In the example shown in Figure 10, we say that the local branch *origin/master* tracks branch *master* in the remote repository. This way, when one pushes to a remote repository, the remote branches are updated to point to the correct commits. The result is shown in Figure 11. Notice that both the local and the remote branches are updated to point to the same commit (labeled 5''), which was created in *clone B* as the result of merging branch *origin/master* into branch *master*.

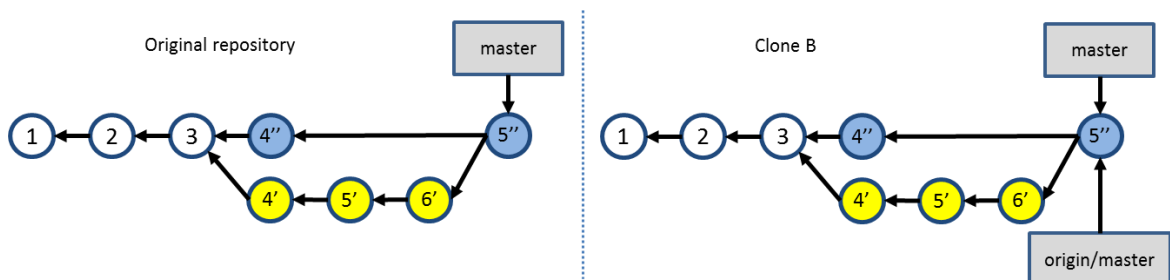


Figure 11 – Result of pushing changes from a tracked branch.

2.3.4 ANONYMOUS BRANCHES

An anonymous branch is a branch created when one checks out a commit that is not referenced by a branch. When this happens, commits are performed in the same way, pointing to their predecessors, with the difference that there is no branch pointing to them (see Figure 12). This way, if one chooses to work with another branch after committing on an anonymous

branch, these commits will be accessible only through their internal identifications generated by the VCS.

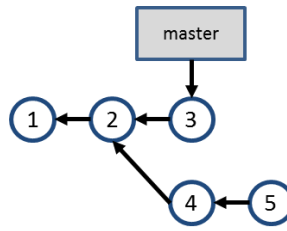


Figure 12 – Anonymous branch

2.4 RELATED WORK

This Section describes some approaches related to awareness or visualization of information stored in VCSs. We used the snowballing search (WEBSTER; WATSON, 2002) to select the approaches, starting with a finite individual population as a seed and looking for these approaches' citations and at approaches that cited them. Our initial seed was based on the referenced papers analyzed by Steinmacher (2012). We also searched at the main academic digital libraries (ACM⁴, IEEE⁵, SpringerLink⁶, and ScienceDirect⁷) and at the industry. We used the following keywords in the search: “revision”, “source code”, “software configuration”, “source control”, and “version control”, combined with “awareness” and “visualization”. The resulting query was (“revision” OR “source code” OR “software configuration” OR “source control” OR “version control”) AND (“awareness” OR “visualization”). We filtered the results found to get only studies that used VCS. The resulting studies were divided into four groups. The first group includes tools that notify commit activities. The second group comprises approaches that give the developer awareness of concurrent changes, sometimes informing them if conflicts were detected. The third group includes approaches that visualize repository information. Finally, the fourth group contains commercial and open source DVCS clients. The next sub-Sections discuss each of these groups.

⁴ <http://dl.acm.org/>

⁵ <http://ieeexplore.ieee.org/>

⁶ <http://link.springer.com/>

⁷ <http://www.sciencedirect.com/>

2.4.1 COMMIT NOTIFICATION

Under the “Commit Notification” group, we found approaches such as *SVN Notifier*⁸, *SCM Notifier*⁹, *Commit Monitor*¹⁰, *SVN Radar*¹¹, *Hg Commit Monitor*¹², and *Elvin* (FITZPATRICK *et al.*, 2006). The primary focus of these approaches is to present to the developer’s new commits performed by other users. Figure 13 shows some of these approaches in action. We can see notifications shown by *SVN Notifier* (Figure 13.a), *SCM Notifier* (Figure 13.b), *Commit Monitor* (Figure 13.c), and *Elvin* (Figure 13.d).

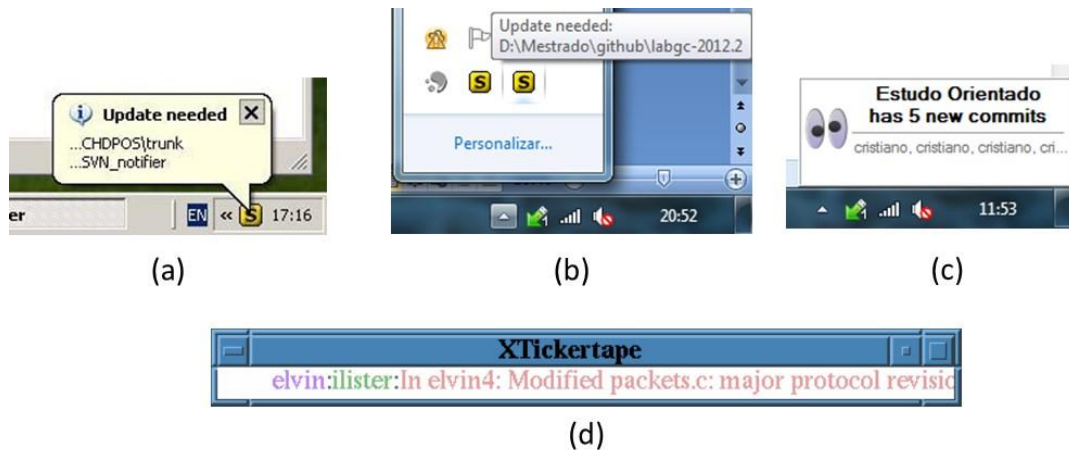


Figure 13 – Commit notification approaches

Except for *SVN Radar* and *Hg Commit Monitor*, all of the approaches found in this group automatically present notifications to the user. While they could be categorized as approaches that provide awareness, they are generally restricted to show only the number of new commits and information related to them (e.g., author, date, log message, and changeset). *Elvin* also uses a publish-and-subscribe event notification system, so that whenever an event occurs in a repository, all the users that subscribed to receive events on that repository are notified and can engage in a chat session to discuss the updates.

Most of these approaches focus on CVCS, except for *SCM Notifier*, and *Hg Commit Monitor*, which support DVCS (Git and Mercurial, respectively), but do not identify related repositories and do not provide information in different levels of details, such as status, branches, and commits.

⁸ <http://svnnotifier.tigris.org/> (2012)

⁹ <https://github.com/pocorall/scm-notifier> (2012)

¹⁰ <http://tools.tortoissvn.net/CommitMonitor.html> (2013)

¹¹ <http://code.google.com/p/svnradar/> (2011)

¹² <http://www.fsmpi.uni-bayreuth.de/~dun3/hg-commit-monitor> (2009)

2.4.2 AWARENESS OF CONCURRENT CHANGES

Many approaches propose increasing the awareness of changes across different team members. *Palantir* (SARMA *et al.*, 2012) shares information about changes to the same files across different workspaces, presenting this information as an eclipse plugin, as shown in Figure 14. The plugin shows the percentage of modification of each file, such as [S:24] in *Payment.java*, which indicates that 24% of the file was modified. It also indicates if there is a direct conflict (marked in the top left with a blue triangle) or an indirect conflict (red triangle in the top right). Direct conflicts are those that occur when there are changes on the same line in different workspaces. Indirect conflicts are those that occur in different lines, for example, when someone changes the signature of a method and the call to that method is not changed to incorporate the change. Although *Palantir* provides awareness regarding modified files, it works only with CVCSSs. It also does not provide views with different levels of detail (e.g. topology, branches, commits).

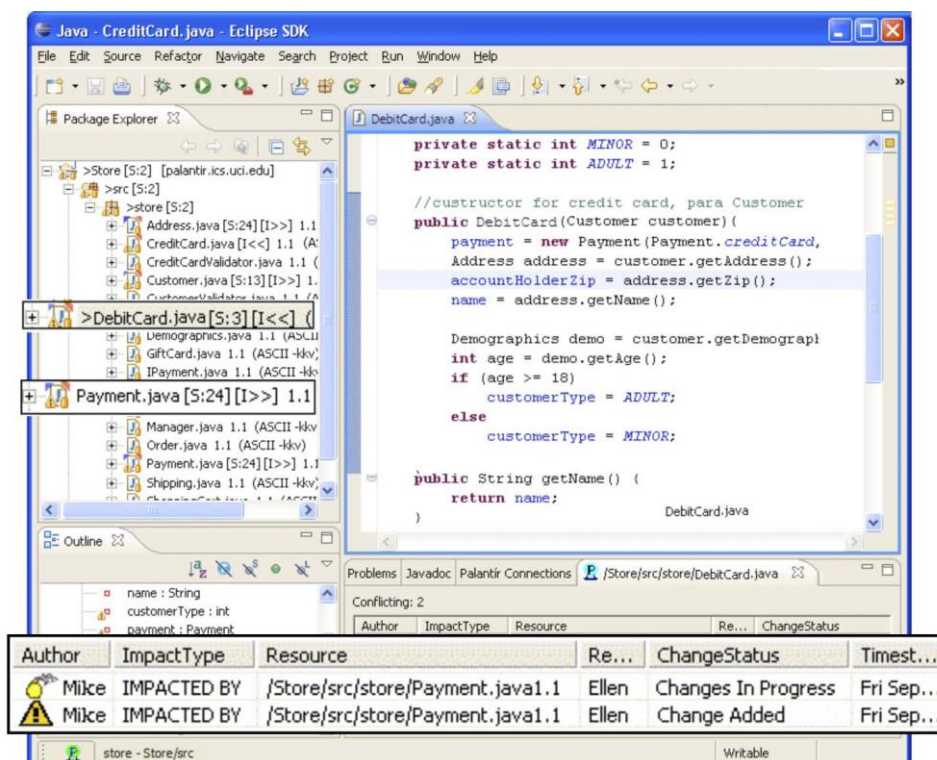
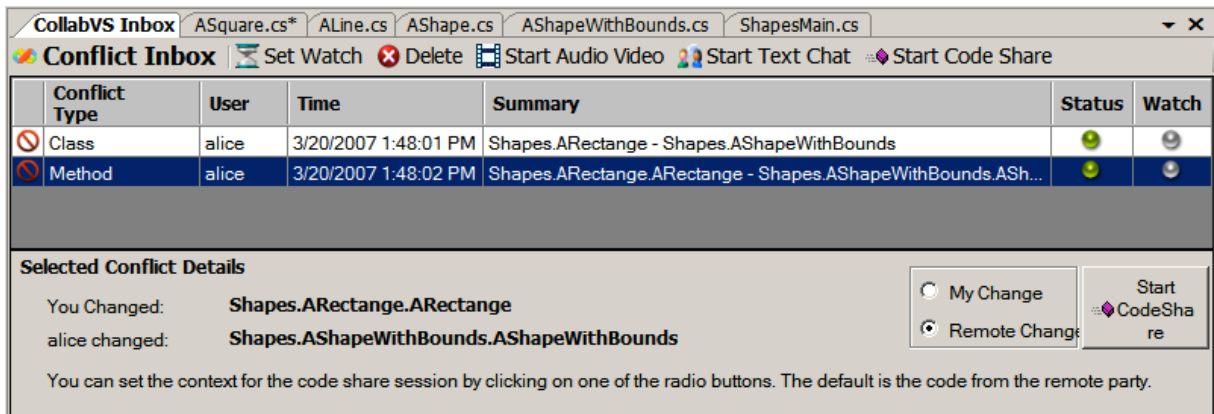


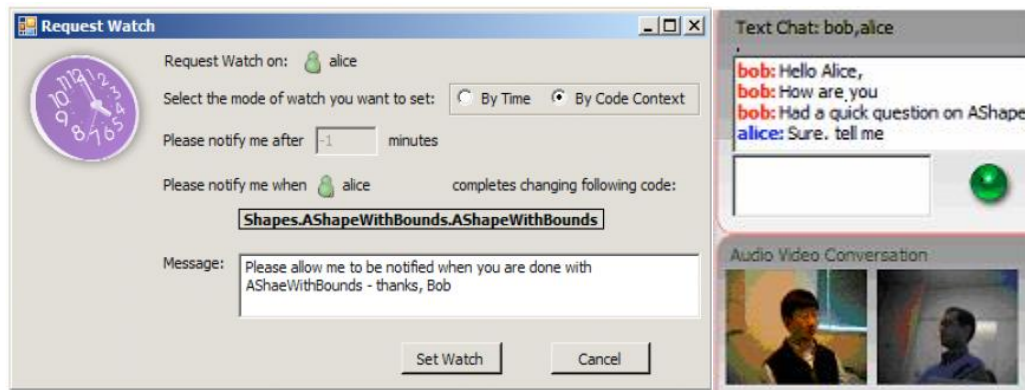
Figure 14 – Eclipse window with Palantir plugin (SARMA *et al.*, 2012)

CollabVS (DEWAN; HEGDE, 2007) is a Visual Studio plugin that identifies conflicts while code is being edited (Figure 15.a), similarly to *Palantir*. The developer can choose between just adding a watch to be notified when their collaborator finishes editing the conflicting code (Figure 15.b) and establishing a communication session with the other author

(Figure 15.c). It also does not work with DVCSs and does not show information in different levels of detail.



(a)



(b)

(c)

Figure 15 – CollabVS snippets (DEWAN; HEGDE, 2007)

FASTDash (BIEHL *et al.*, 2007) does not detect conflicts directly, but provides awareness of potential conflicts, such as two programmers editing the same region of the same source file (direct conflicts), in repositories stored in Team Foundation Server¹³. The notifications can be presented as a plugin to Visual Studio. Figure 16 shows *FASTDash* plugin in action. *FastDASH* works with DVCSs (Team Foundation Server), but it neither shows information in different levels of detail nor deals with repositories that pull updates from more than one peer.

¹³ <http://www.visualstudio.com/en-us/products/tfs-overview-vs.aspx> (2013)

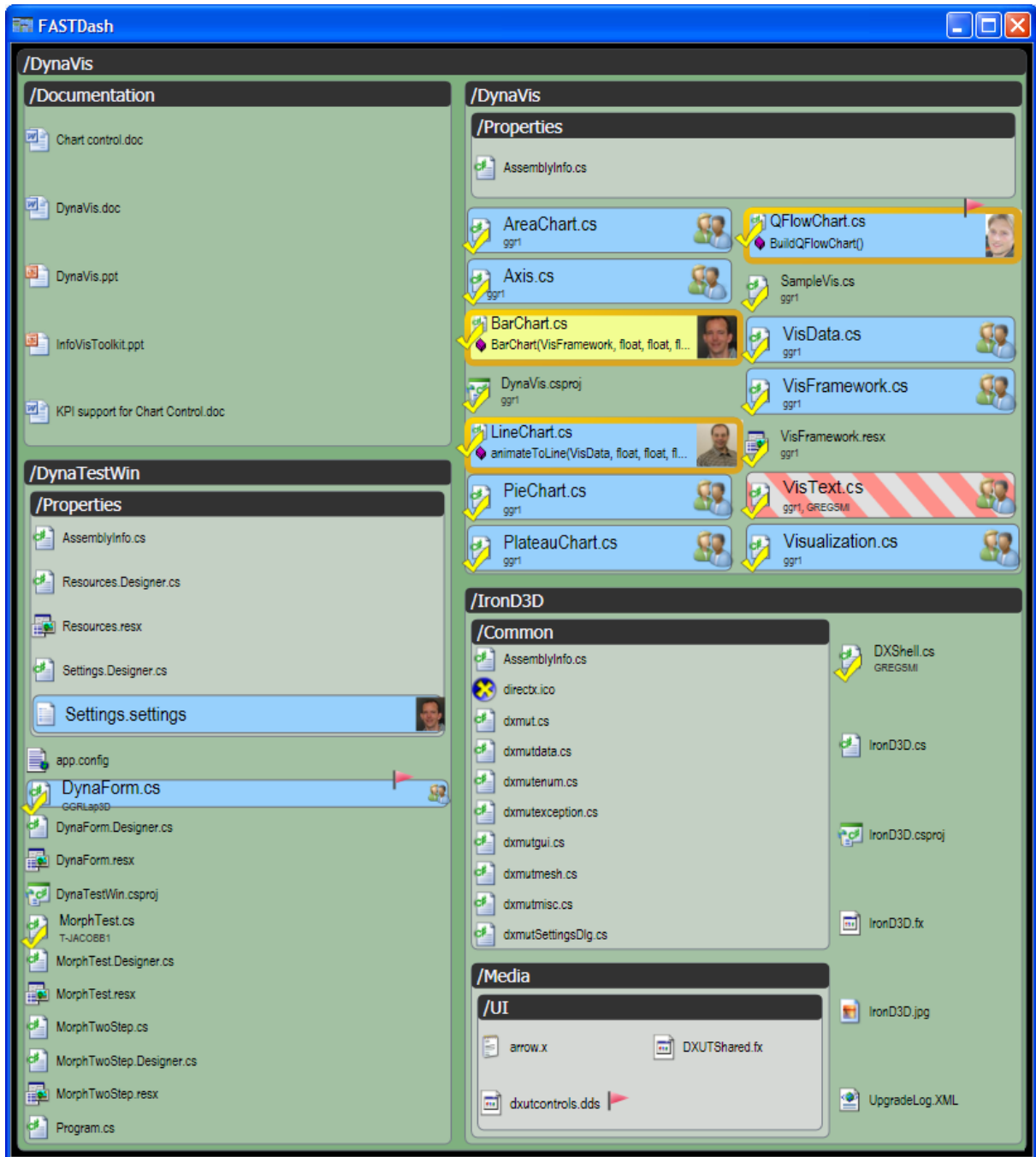
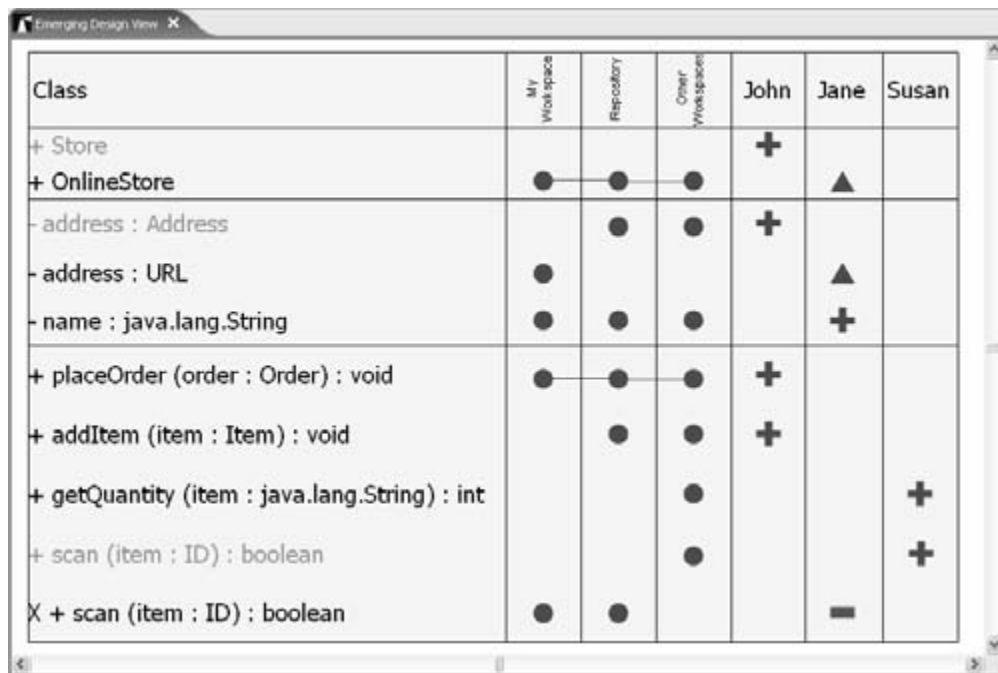


Figure 16 – FASTDash Visualization Runtime (BIEHL *et al.*, 2007)

Lighthouse (DA SILVA *et al.*, 2006) is an Eclipse plug-in that monitors workspaces for changes as soon as they are made, in order to address conflicts sooner. It presents an updated design (Figure 17) indicating where the changes have been made and by whom, as well as if these changes have already been propagated to the repository and to other workspaces. The granularity of information shown is any Java element (class, interface, attribute or method). *Lighthouse* is able to show that an element does not exist locally, but exists in some peer (similarly as *DyeVC* does), like the *Store* class in Figure 17, that is marked as added by John,

but does not exist at the developer's workspace. Again, this approach does not work with DVCSs and does not show different levels of detail.



Class	My Workspace	Repository	Other Workspaces	John	Jane	Susan
+ Store				+		
+ OnlineStore	●	●	●		▲	
- address : Address		●	●	+		
- address : URL	●				▲	
- name : java.lang.String	●	●	●		+	
+ placeOrder (order : Order) : void	●	●	●	+		
+ addItem (item : Item) : void		●	●	+		
+ getQuantity (item : java.lang.String) : int			●			+
+ scan (item : ID) : boolean			●			+
X + scan (item : ID) : boolean	●	●			-	

Figure 17 – Lighthouse plugin on Eclipse (DA SILVA et al., 2006)

WeCode (GUIMARÃES; SILVA, 2012) is an eclipse plugin for real-time integration of changes, which is accomplished by means of continuously merging the working copy with work being done by other team members. Figure 18 shows the plugin in action. The Team view (3) shows the members of the team, along with their changes. The Package Explorer (1) and the Source Code view (2) were adapted to indicate existing conflicts. The Team Merge view (4) shows the conflicts that arise during continuous merge. By the time we wrote this work, *WeCode* supported only CVCS, but the authors had plans to support other VCSs, like Git and Mercurial. Similarly to the approaches discussed previously, this approach does not show different levels of information detail.

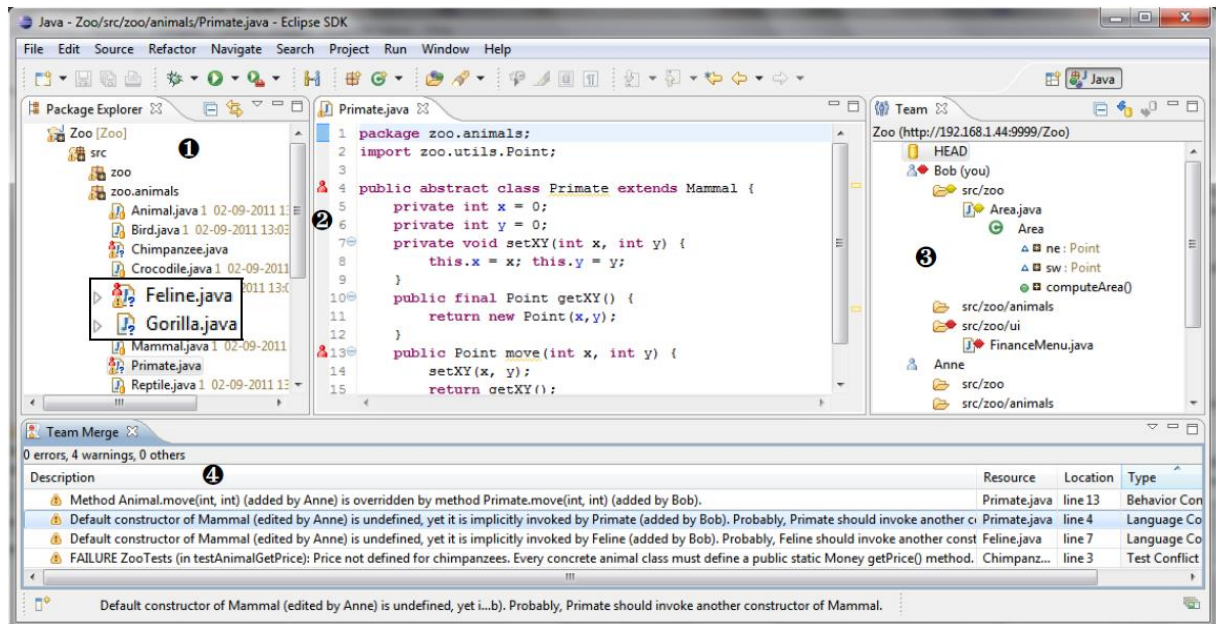


Figure 18 – WeCode continuous merging (GUIMARÃES; SILVA, 2012)

Crystal (BRUN *et al.*, 2011) performs continuously merges between pairs of repositories, compiling and testing the merged code, resulting in “merge failure”, “compilation failure”, “tests failure”, or “tests passed” (Figure 19).

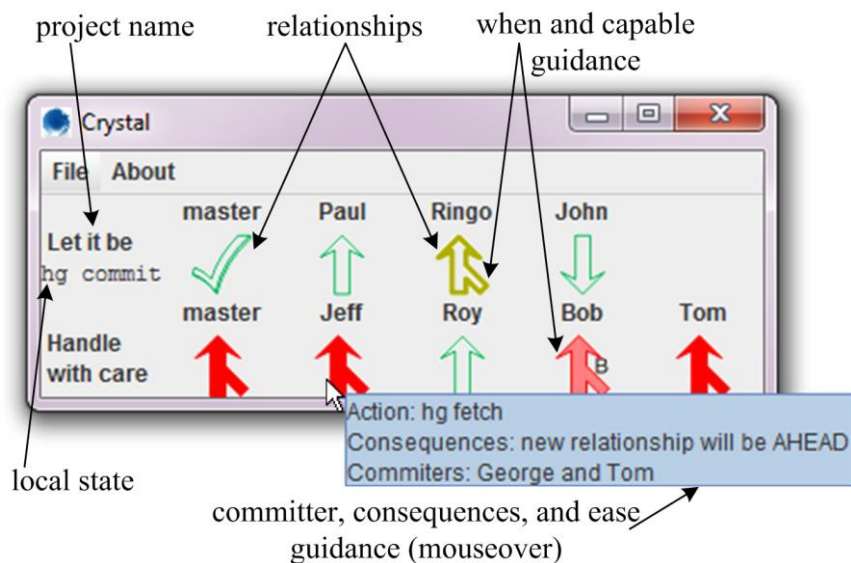


Figure 19 – Crystal snapshot (BRUN *et al.*, 2011)

Crystal works with DVCSs (Git and Mercurial), but it does not automatically find related repositories, demanding that the user points out every repository they want to compare. Besides that, it does not show information in different levels of detail (e.g., repositories, branches, and commits). It also does not deal with more than one branch, as each different branch to be analyzed has to be manually pointed out in the tool.

2.4.3 REPOSITORY VISUALIZATION

Repository visualization consists in taking the information stored in VCSs and presenting it in a visual way. A number of approaches exist that propose different visualizations with different focuses. *Gevol* (COLLBERG *et al.*, 2003) focuses on program structures. It extracts information about a Java program stored in a CVS version control system and displays it using a temporal graph visualizer. Figure 20 shows an example of a call graph, where nodes start out red, and in the case that no changes exist they turn purple and then blue. When a change is performed, the affected nodes turn red and the cycle starts again.

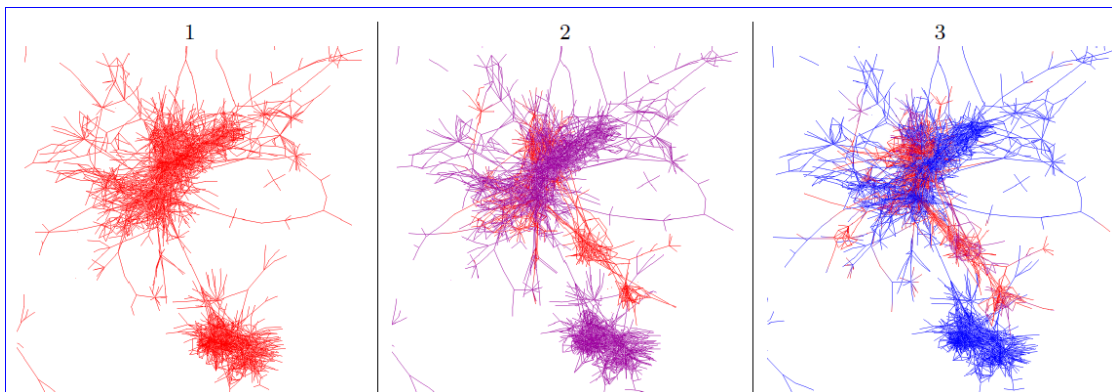


Figure 20 – Gevol: focus on program structures (COLLBERG *et al.*, 2003)

The Evolution Matrix (LANZA, 2001) focuses on classes and their evolution within a software system. Figure 21 shows a snapshot of the approach. The columns represent different versions of the software and the rows represent different classes. The size of the lines represents the size of the classes. This way, it is possible to see if the system is growing, shrinking or becoming stagnated.

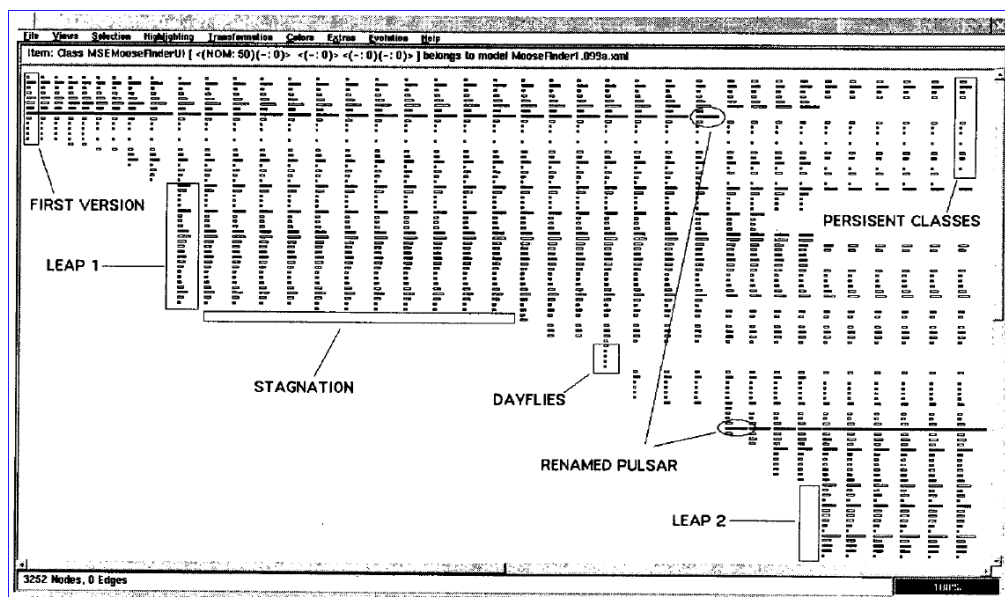


Figure 21 – The Evolution Matrix: focus on classes (LANZA, 2001)

CVSScan (VOINEA *et al.*, 2005) is a line-oriented approach that shows how a given file changed over the time, through different versions. Figure 22 shows a snapshot of the approach for a file. Each column represents a version and the rows represent lines of the file. The color scheme indicates if the line is the same (green), inserted (blue), deleted (red), or modified (yellow). Different color schemes can be chosen to depict, for example, the author of the last change in each line or the type of construct that the line represents (file reference, block, or comment).

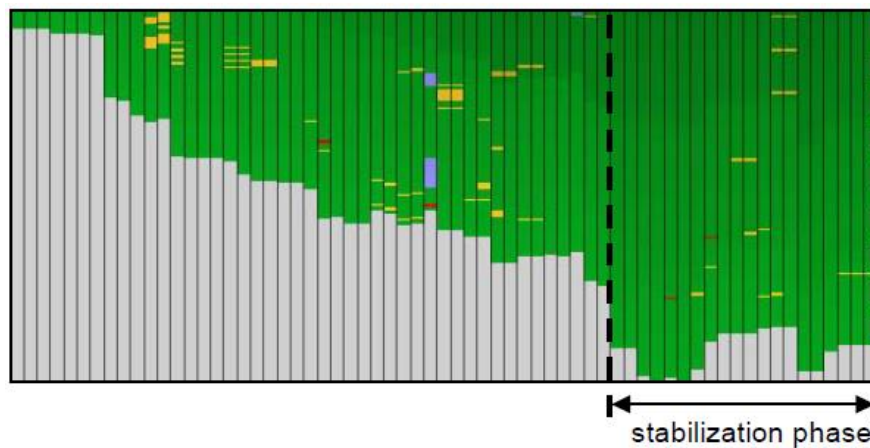


Figure 22 – CVSScan: focus on lines (VOINEA *et al.*, 2005)

Lifesource (GILBERT; KARAHALIOS, 2006) focus on showing how authors are contributing to a project. Figure 23 presents a visualization named *CodeSaw*, which shows the contributions along one year. Each colored line represents a different author and has two axes: the upper axis represents code contributions and the lower axis represents e-mail contributions to project mailing list.

Polvo (SANTOS; MURTA, 2012) is an approach that establishes metrics that assist in determining the merge effort between branches, by quantifying merging complexity between involving Subversion branches. Figure 24 presents *Polvo* showing the metric *Precision by Amount of Artifacts*, which calculates the relation between the number of identical artifacts over the total number of artifacts in the branch.

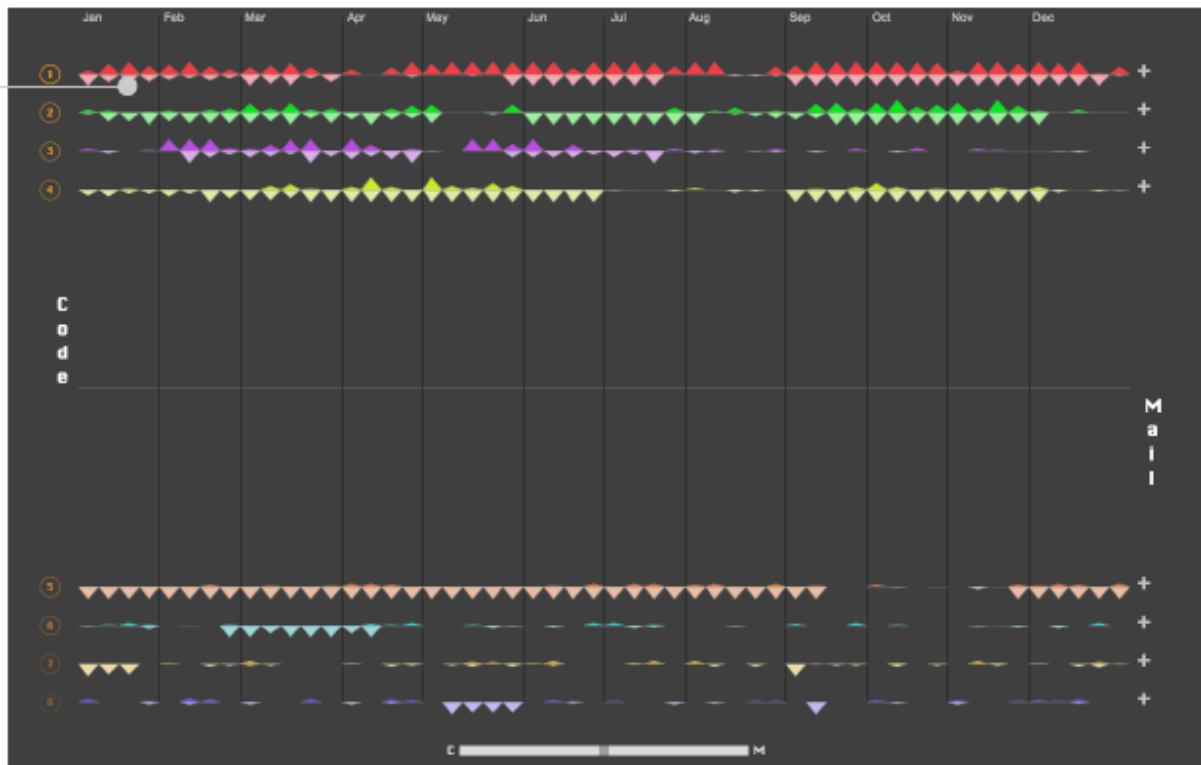


Figure 23 – LifeSource: focus on authors (GILBERT; KARAHALIOS, 2006)

Project Overview

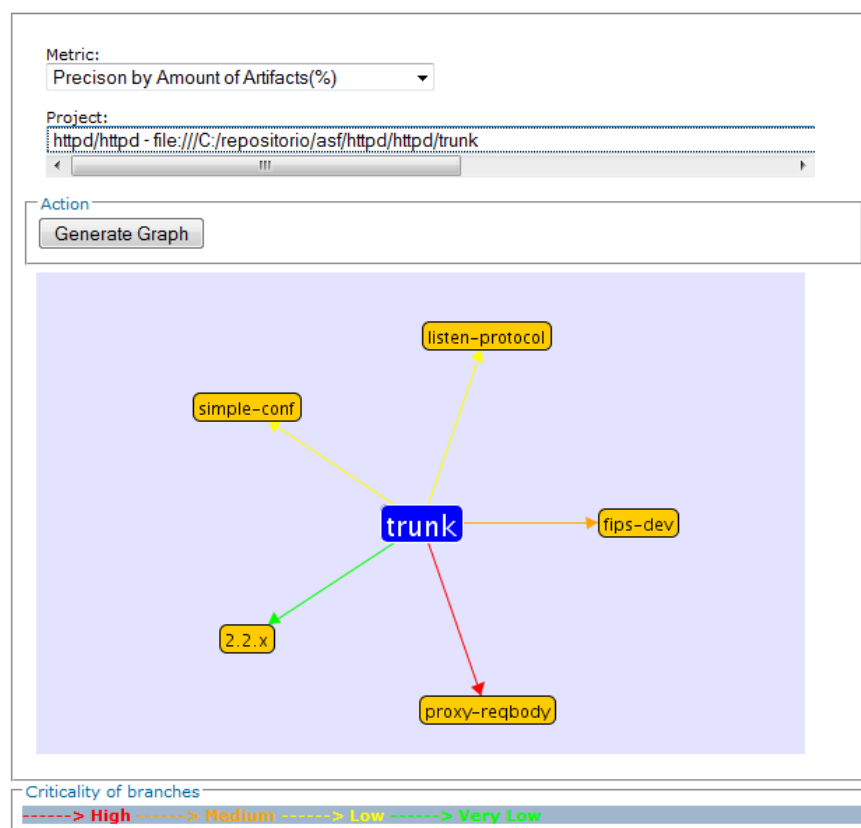


Figure 24 – Polvo: focus on branches (SANTOS; MURTA, 2012)

VisGi (ELSEN, 2013), *Visugit* (HOZUMI, 2010), and *GitHub's Network Graph* (PRESTON-WERNER, 2008) were designed to incorporate information about the different branches of a repository, using graphs to render the information. Figure 25 shows three different repositories represented by *VisGi*, which uses a horizontal layout that represents time in a left-right direction. *VisGi* shows branches as solid black dots and branches intersections as dark gray dots.

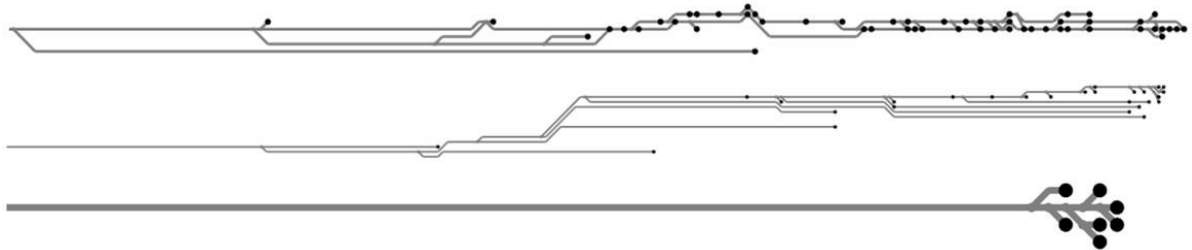


Figure 25 – VisGi: focus on branches (ELSEN, 2013)

Figure 26 presents a snapshot of *Visugit*, which shows the commits in a repository, depicting the existing branches (e.g., github/master and master).

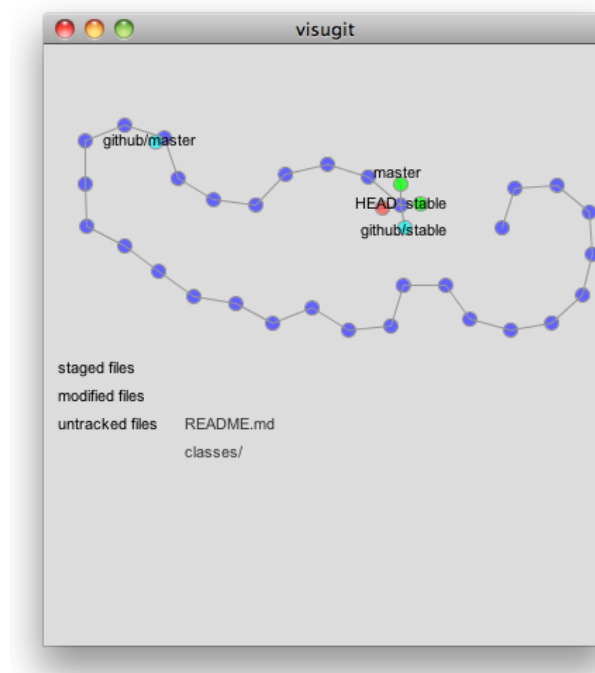


Figure 26 – Visugit: focus on branches (HOZUMI, 2010)

Figure 27 presents *GitHub's Network Graph*, which shows all the commits associated with a repository, along with the information about branches (shown in the tag markers). The graph is drawn from the perspective of a given user (which is called *root user* – in this case, *mojombo*) and all the commits that exist in *mojombo's* repository are drawn across his name.

For other users, commits that exist in their repositories but not in *mojombo*'s are drawn across their names.

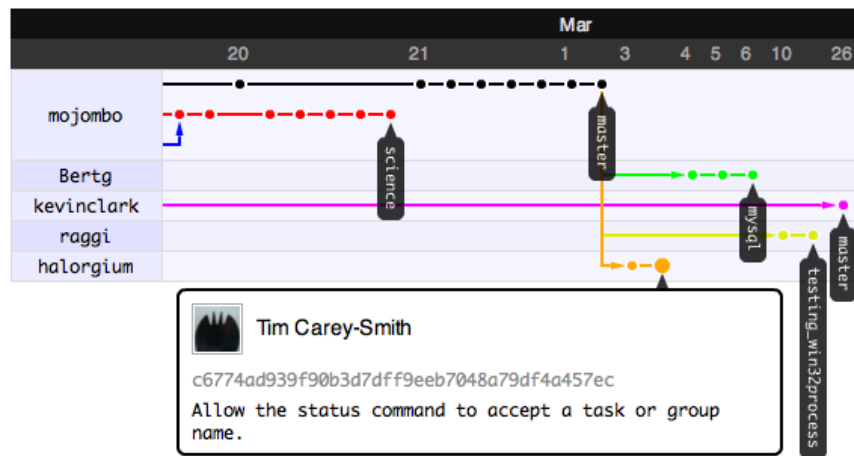


Figure 27 – GitHub's Network Graph: focus on branches (PRESTON-WERNER, 2008)

While all of the approaches presented in this category focus on visualizing repository information, only the last three work with DVCSs (*VisGi*, *Visugit*, and *GitHub's Network Graph*). They visualize the repository history (*commits*), but they look only at a local repository, not showing, for example, where a given commit can be found. At first sight, the work of Preston-Werner seems to address this topic, but it actually does not address it entirely. Giving a repository *rep*, *GitHub's Network Graph* allow one to see commits that a *rep*'s peer has and *rep* does not, but it might be the case that a commit exists only locally in *rep* and does not exist anywhere else (*rep* did not pushed it yet). This scenario cannot be seen in this visualization. In addition, these approaches do not show the dependencies among DVCS clones and do not provide information in different levels of detail.

2.4.4 DVCS CLIENTS

It is worth noticing that there is a number of commercial / open source DVCS clients, which allows one to execute operations on repositories / clones (push, pull, checkout, commit, etc.) and also visualizing the repository history, i.e., the commits, along with their attributes (comment, date, affected files, committer, etc.). For example, for Git repositories, some clients include *gitk*¹⁴, *TortoiseGit*¹⁵, *EGit for Eclipse*¹⁶, and *SourceTree*¹⁷. There is an extensive list of

¹⁴ <http://git-scm.com/docs/gitk>

¹⁵ <https://code.google.com/p/tortoisegit/>

¹⁶ <http://eclipse.org/egit/>

¹⁷ <http://www.sourcetreeapp.com/>

these tools available in the Git Wiki¹⁸. Figure 28 and Figure 29 present screenshots taken from *gitk* and *TortoiseGit*, respectively.

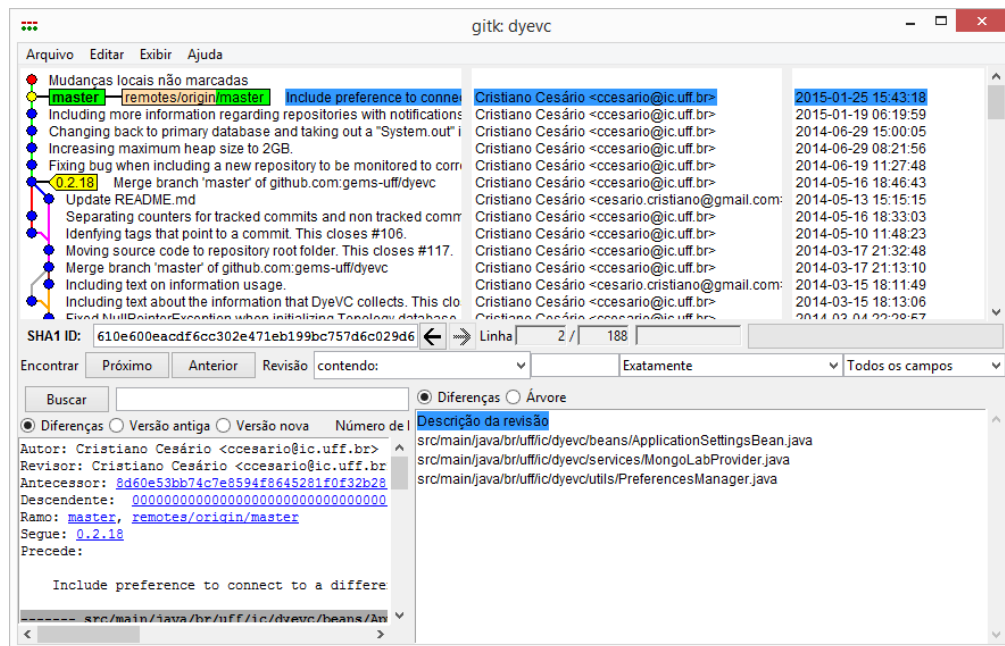


Figure 28 – gitk client

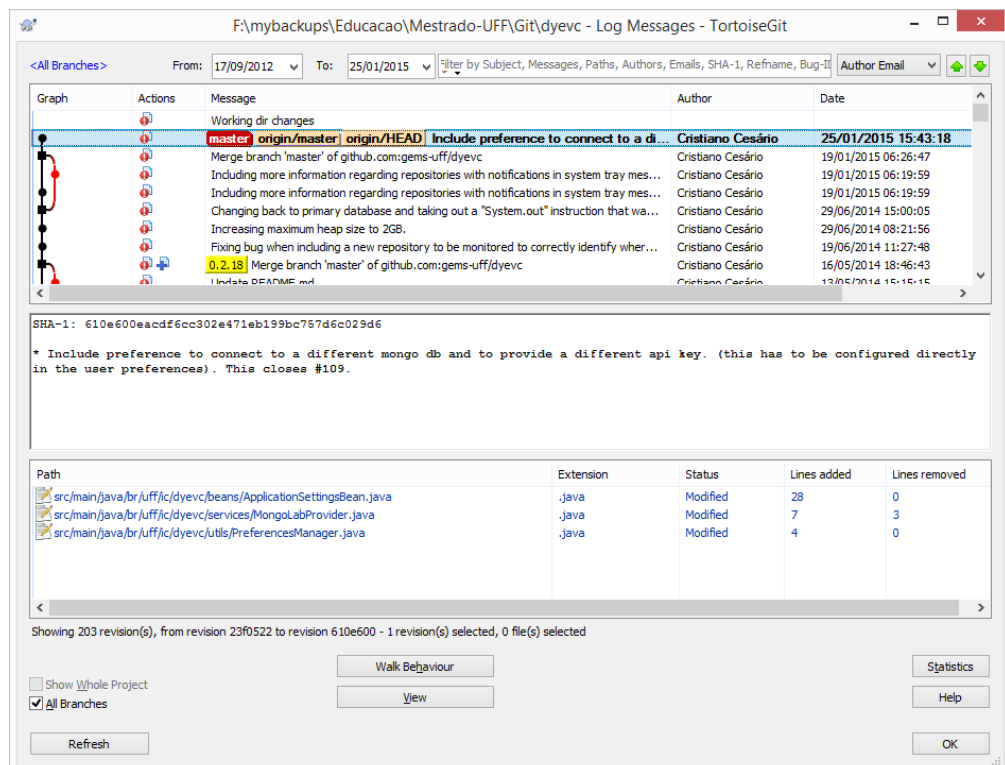


Figure 29 – TortoiseGit client

¹⁸ <https://git.wiki.kernel.org/index.php/InterfacesFrontendsAndTools>

The data about commits shown by these tools varies, but generally involves the committer name, message, date, affected files, and a visual representation of the history, which can be seen on the left part of Figure 28 and Figure 29. These tools, though, have no knowledge regarding peers. For this reason, these tools do not present commits from other clones and do not include information about where each commit can be found.

2.5 FINAL CONSIDERATIONS

This chapter presented some core concepts regarding DVCS, which are essential to understand the remaining of this work. We also presented four groups of approaches related with this work: Commit Notification, Awareness of Concurrent Changes, Repository Visualization, and DVCS clients. Although some approaches shown work with DVCS, none of them provides information in different levels of detail, nor shows information regarding several different clones, nor shows the dependencies among several DVCS clones.

CHAPTER 3 – APPROACH

3.1 INTRODUCTION

According to Diehl (2007), software visualization can be separated into three aspects: structure, behavior, and evolution. DyeVC relates primarily with the evolution aspect, more specifically with studies that aim at improving the awareness for people that work with distributed software development. A recent work by Steinmacher *et al.* (2012) presents a systematic review of awareness studies and classify them according to the Awareness Framework (GUTWIN *et al.*, 1996) and according to the 3C Collaboration Model (FUKS *et al.*, 2007). The classification is not exclusive, i.e., a given tool can present elements of different awareness types. According to Gutwin *et al.* (1996), DyeVC can be classified as a “Workspace Awareness” approach and according to Fuks (2007), DyeVC fits into the “Coordination” and “Cooperation” categories.

As we have discussed in Section 2.2, DVCSs lead to a number of repository clones that may communicate with each other, receiving or sending updates. This operating mode resembles a peer-to-peer network topology (SCHOLLMEIER, 2001), where there are processing units and the flows between them through predefined connection paths. Whereas there are several approaches to discover such network topologies (DONG; GANG, 2012; LI, H. *et al.*, 2009; LI, M. *et al.*, 2013; UZAIR *et al.*, 2007; YAN, 2012; YONG *et al.*, 2010), to number a few, there is no corresponding approach to deal with DVCS, as discussed in Chapter 2.

The DyeVC approach (CESARIO; MURTA, 2013) came to fill this gap in supporting DVCS usage. The goal of DyeVC is two-fold. First, DyeVC should work as a non-obtrusive awareness tool to increase the developer knowledge on what is going on around their repository and the repositories of their teammates. Second, DyeVC should enable repository administrators and/or managers to visualize how the several existing repositories of a project interact with each other.

This chapter explains the DyeVC approach, which consists of series of visualizations built upon DVCS environments. These visualizations provide different levels of detail that allow those involved in projects using DVCS to:

- Receive notifications in the system tray bar whenever changes are detected in related peers (i.e., clones from where a repository pulls from or pushes to);

- Visualize all known clones of a project, and their interdependencies (i.e., which are the existing communication paths among them);
- Visualize information regarding tracked branches, and their status compared to their corresponding branches in the original repository;
- Visualize a repository history that contains all commits in the topology, even those that do not exist locally.

Besides these visualizations, DyeVC has also a mechanism to gather information from a set of clones, processing this information and storing it to allow its presentation in the aforementioned visualizations.

This chapter is organized as follows: Section 3.2 explains the data model used to store the information that our approach gathers and how this information is gathered from DVCSs. Section 3.3 shows how this information is presented using different levels of detail. Section 3.4 discusses details regarding the information gathering process. Section 3.5 presents the technologies used in the prototype implementation. Lastly, Section 3.6 presents the final considerations of this chapter.

3.2 INFORMATION GATHERING

DyeVC continuously gathers information from a group of interrelated clones, starting from clones registered by the users. As shown in Figure 30, data is gathered by DyeVC instances running at each user machine and is stored in a central document database. This way, information from one DyeVC instance is made available to every other instance in the topology. For each registered clone *rep*, DyeVC transparently creates a clone *rep'* in the user's home folder. *Rep'* is a working copy used to perform fetches from all of the peers that *rep* communicates with. We use a working copy because performing fetches on the monitored repository would change its structure, bringing new commits to the repository without the user being aware of it. DyeVC needs to perform fetches to analyze branches status (see more in Section 3.3.3).

DyeVC gathers information not only from the registered clones in the user's machine, but also from its peers, which are the clones that a given clone communicates with. Since there is a communication path between a registered clone and its peers (in order to push and pull data), we are able to analyze the commits that exist in these peers. This allows us to present a broader topology visualization that contains not only registered clones, but also those that have

a push or pull relationship with them. Details on how data is gathered are explained in Section 3.4.

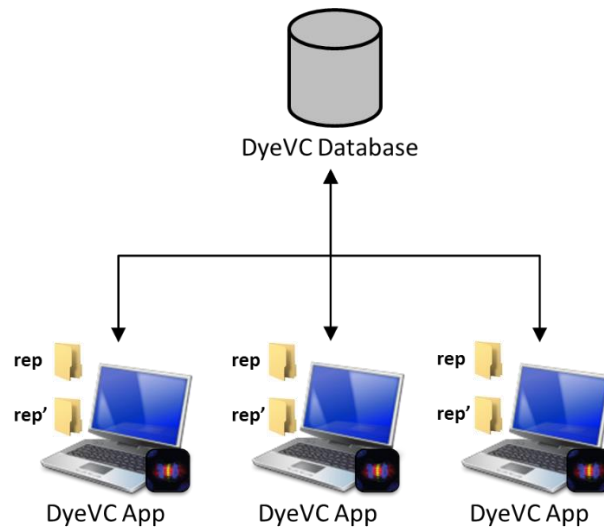


Figure 30 – How DyeVC gathers information

Figure 31 shows how DyeVC discovers the topology from the nodes where it is running and the registered clones. Here, blue nodes represent registered clones, where DyeVC is running, yellow nodes represent known clones located at nodes where DyeVC is not running, dashed nodes and dashed lines represent clones and communication paths that are not yet known. Suppose a scenario where the existing clones and interdependencies are shown in Figure 31.a. After installing DyeVC and registering clone 3, DyeVC finds out that this clone communicates with clones 1, 2, and 4 (either by pushing to or pulling from them), which is shown on Figure 31.b. Later on, clone 4 is registered and clone 5 is included as a known clone in the topology (Figure 31.c). Clone 6 is the next to be registered, allowing DyeVC to discover that clone 7 also exists, as well as the communication between clone 6 and clone 1, which was already a known clone (Figure 31.d). Suppose that no more clones are registered. The known topology that will be shown will be that of Figure 31.e. Notice that, although only clones 3, 4, and 6 were registered, DyeVC is also aware of the existence of clones 1, 2, 5, and 7. Only clone 8 will not be known, as well as some communication paths between clones that were not registered (1-2, 1-5, 1-8, and 7-8).

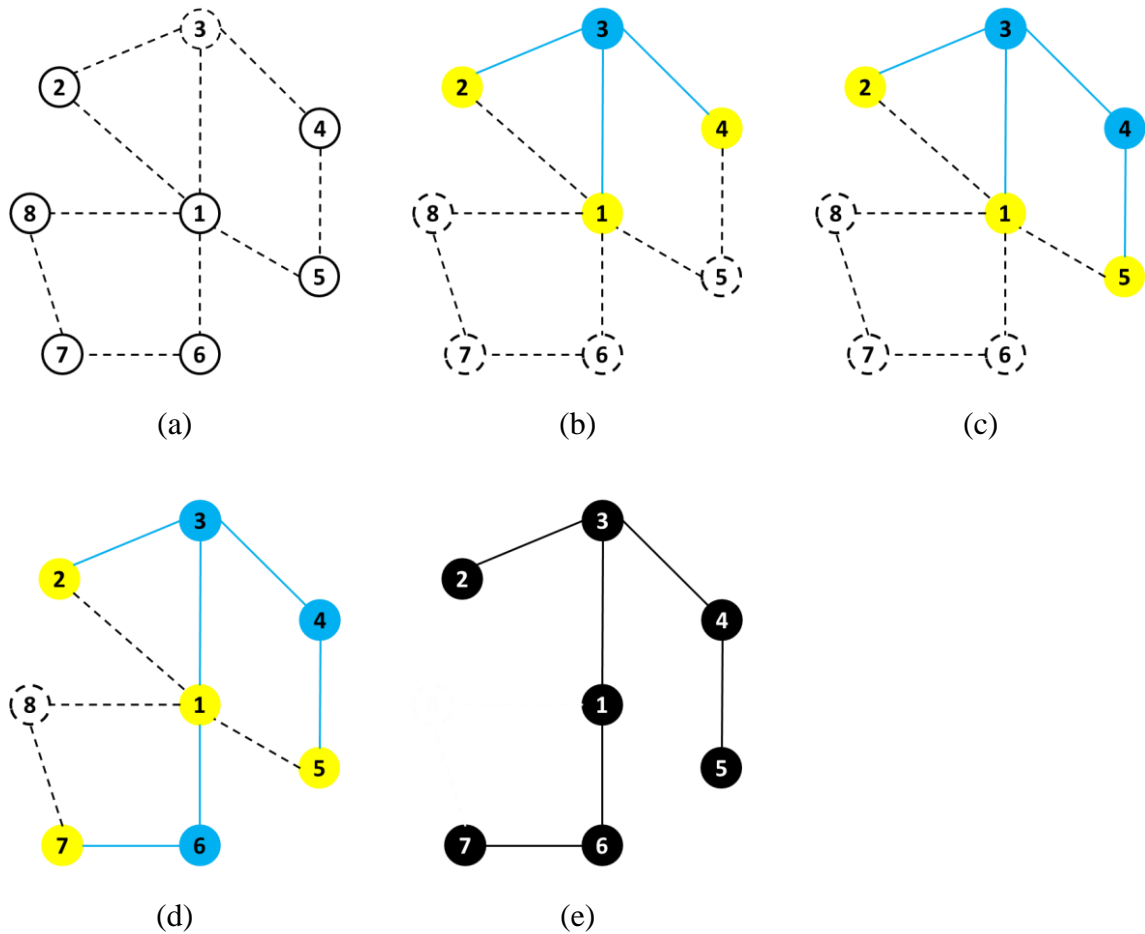


Figure 31 – DyeVC discovering the topology

DyeVC finds out related clones by looking at the remote repositories, which are registered in Git's *config* file of each clone. Figure 32 shows an example of this configuration, taken from a local clone of the *dyevc* project, where there is a remote named *origin*, which is located at *github.com/gems-uff/dyevc*.

```
[remote "origin"]
  fetch = +refs/heads/*:refs/remotes/origin/*
  url = git@github.com:gems-uff/dyevc.git
```

Figure 32 – Remote repository configuration in Git's *config* file

The data stored at the central database follows the model presented in Figure 33. A *Project* groups all repository clones of the same system, and each project is identified by a project name. Repository clones are stored as *RepositoryInfo* and are identified by an id and a meaningful clone name provided by the user. A *RepositoryInfo* has a list of clones to which it pushes to and a list of clones from which it pulls from. These lists are represented respectively by the self-associations *pushesTo* and *pullsFrom*. We also store the list of DyeVC instances that references the clone, in order to remove from the topology clones that are no longer referenced

(for example, a central repository may be referenced by several DyeVC instances). Finally, we store the hostname where the clone resides, as well as its path, which can be either an operating system path or a URL. DyeVC supports data transferring using the same set of protocols supported by Git, i.e., local (*file://*), http/s (*http://* or *https://*), secure shell (*ssh://*), and git (*git://*).

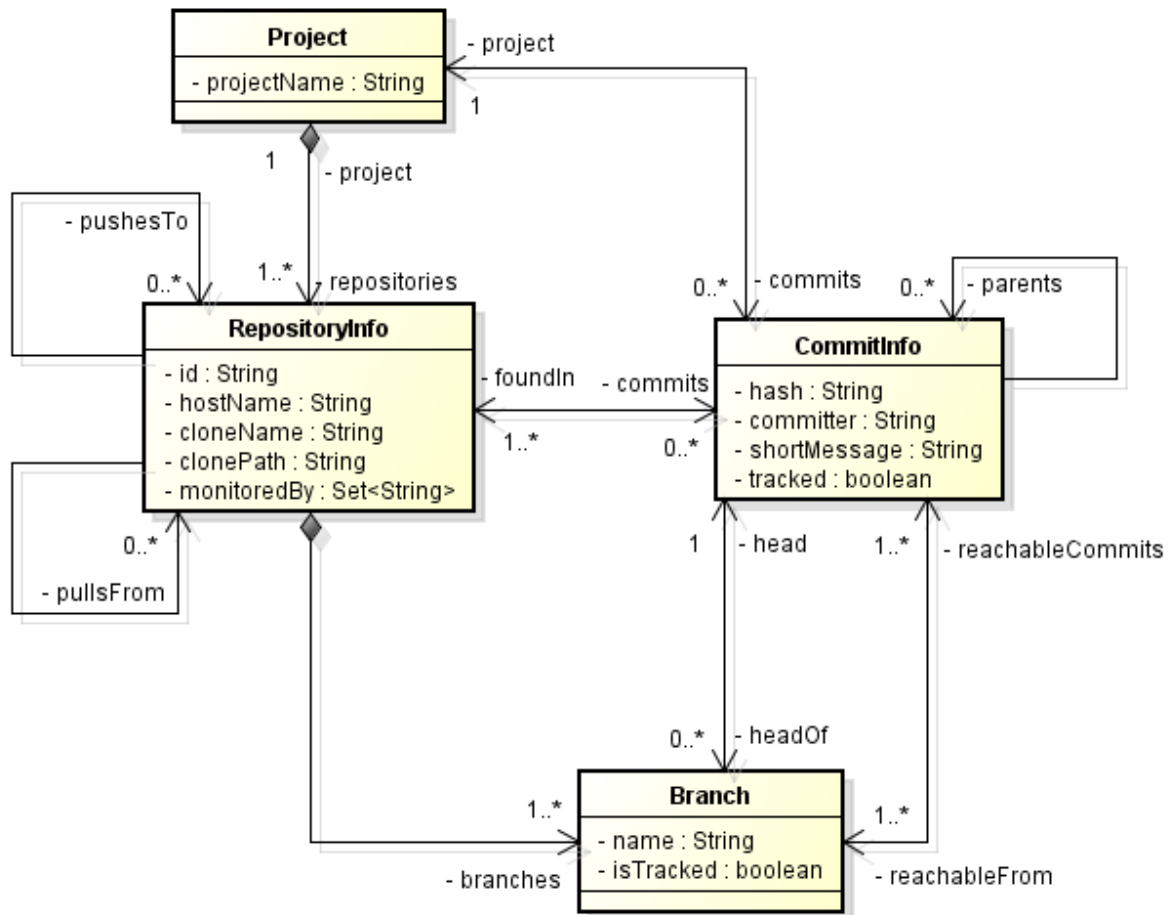


Figure 33 – Model used to store topology data

Another element in Figure 33 is the *Branch*. Branches are part of a *RepositoryInfo*. A *Branch* instance has a name and a Boolean attribute *isTracked*, which is true if the branch tracks a remote branch. A *RepositoryInfo* may have one or many branches (it must have at least one branch, which is the main one). A *Branch* has two associations with *CommitInfo*: through the first association, a *Branch* knows which commit is its head and, conversely, a commit knows which branches point to it as a head. The second association represents which commits are reachable from a given branch and, conversely, the branches from which the commit is reachable.

The finer grain of information is the *CommitInfo*, which represents each commit in the topology. A commit is identified by a hash code and it refers to its parents (except for the first

commit in the repository, which does not have any parent). As each commit may not exist in all clones of the topology, we store the list of clones where each commit can be found (*foundIn* association). We also store the committer, the commit message, and the information whether the commits belongs to tracked branches or to non-tracked branches.

3.3 INFORMATION VISUALIZATION

The visualization of information gathered by DyeVC is classified into four different levels of detail:

- Level 1 presents high-level notifications about the registered repositories.
- Level 2 presents the whole topology of a given project.
- Level 3 zooms into the branches of the repository, to see the status of each local branch that tracks a remote branch.
- Level 4 zooms into the commits of the repository, to see a visual log with information about each commit.

The following Sections discuss each of these levels.

3.3.1 LEVEL 1: NOTIFICATIONS

In Level 1, our approach periodically monitors the registered repositories and presents notifications whenever a change is detected in any known peer. The period between subsequent runs is configurable, and the notifications are presented in the system notification area, in a non-obtrusive way, allowing the user to begin investigating what is occurring, if desired. Figure 34 shows an example of this kind of notification, where DyeVC detected changes in two different repositories. The notification shows the repository id, followed by the name given by the user and the project name (system name). Clicking on the balloon allows the user to start investigating what changed.

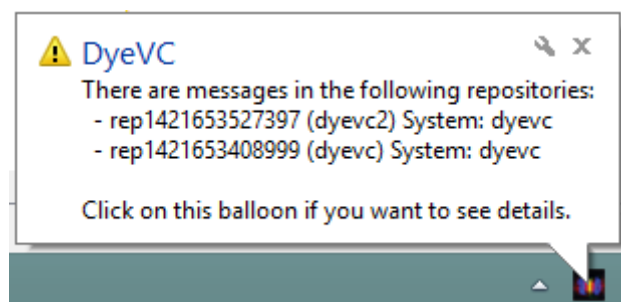


Figure 34 – DyeVC showing notifications in the notification area

3.3.2 LEVEL 2: TOPOLOGY

Aiming at helping answering questions Q1 and Q2 from Section 1.1, we present a topology view showing all repositories for a given project, as depicted in Figure 35, where each node represents a known clone of the project *dyevc* itself, at a given moment. A blue computer represents the current user clone and black computers represent other clones where DyeVC is running. Servers represent central repositories, that do not pull from nor push to any other clone, or clones where DyeVC is not running. The representation is the same for both kinds of nodes because, once DyeVC is not running at a given clone, we cannot infer if the clone pushes to or pulls from anyone else. Thus, it will have empty push and pull lists and will be understood as a server.

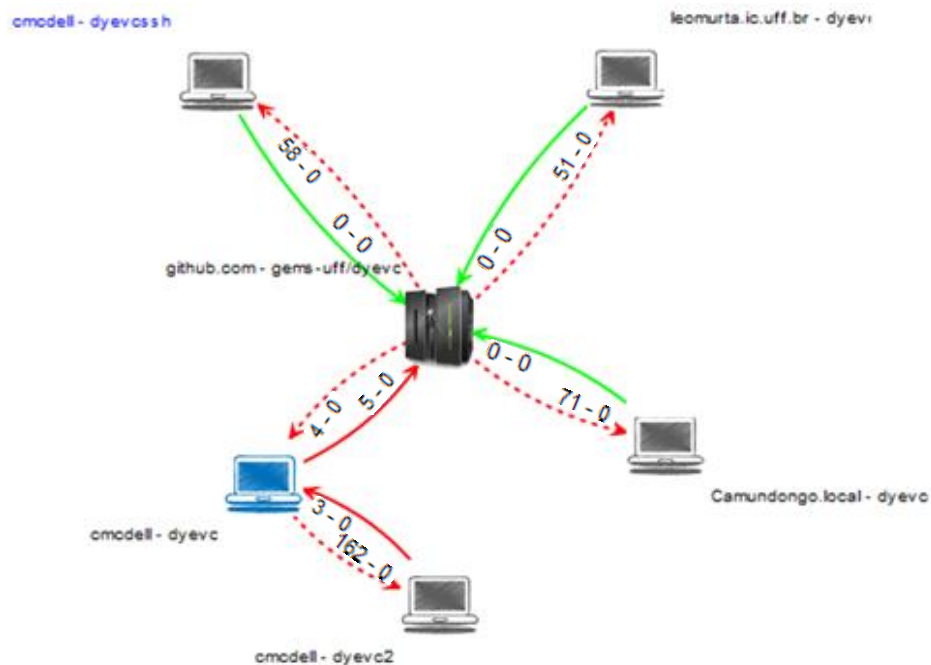


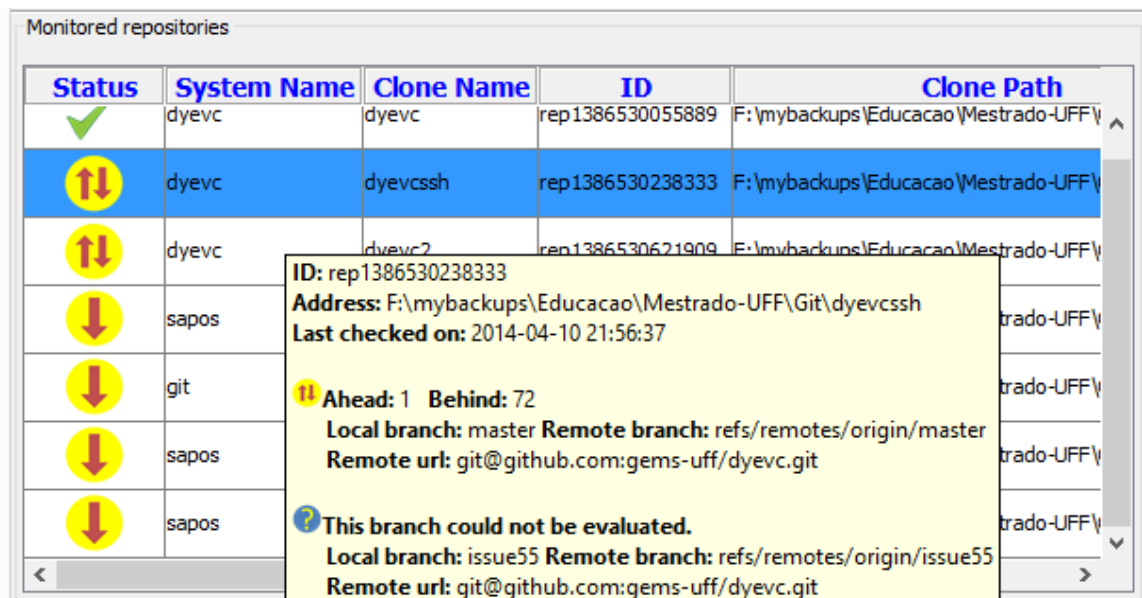
Figure 35 – Topology view of DyeVC project, at a given moment

Each edge in the graph represents a relationship between two repositories. Edges with a continuous stroke mean that the source clone pushes to the destination clone. Edges with a dashed stroke mean that the destination clone pulls from the source clone. The edge labels show two numbers separated by a dash. The first number represents how many commits in tracked branches of the source clone are missing in the destination clone. The second number represents how many commits in non-tracked branches of the source clone are missing in the destination clone. The edge colors are used to represent the synchronization status: green edges mean that both clones are synchronized (i.e., both clones have the same set of commits), whereas red edges mean that the pair is not synchronized.

For example, it is possible to observe in Figure 35 that the current user clone (blue computer) is hosted at *cmcdell* and it is named *dyevc*. This clone pulls from *gems-uff/dyevc*, which is located at *github.com*, and there are four tracked commits ready to be pulled. It also pushes to the same peer, having five tracked commits ready to be pushed.

3.3.3 LEVEL 3: TRACKED BRANCHES

To answer question Q3 from Section 1.1, our DyeVC's main screen (see Figure 36) shows Level 3 information, allowing one to depict the status of each tracked branch between registered repositories and their peers. This information is complemented with that of Level 4, shown in Section 3.3.4.



Status	System Name	Clone Name	ID	Clone Path
✓	dyevc	dyevc	rep1386530055889	F:\mybackups\Educacao\Mestrado-UFF\
↕	dyevc	dyevcssh	rep1386530238333	F:\mybackups\Educacao\Mestrado-UFF\
↕	dyevc	dyevc2	rep1386530621909	F:\mybackups\Educacao\Mestrado-UFF\
↓	sapos			trado-UFF\
↓	git			trado-UFF\
↓	sapos			trado-UFF\
↓	sapos			trado-UFF\

ID: rep1386530238333
Address: F:\mybackups\Educacao\Mestrado-UFF\Git\dyevcssh
Last checked on: 2014-04-10 21:56:37

⚠ **Ahead: 1 Behind: 72**
Local branch: master **Remote branch:** refs/remotes/origin/master
Remote url: git@github.com:gems-uff/dyevc.git

❓ **This branch could not be evaluated.**
Local branch: issue55 **Remote branch:** refs/remotes/origin/issue55
Remote url: git@github.com:gems-uff/dyevc.git

Figure 36 – DyeVC main screen

The status evaluation considers the existing commits in each repository individually. Table 1 shows the possible states presented by DyeVC. Due to the nature of DVCS, old data is almost never deleted and commits are cumulative. Only in rare situations, such as removal of sensitive data mistakenly committed to the repository, commits are deleted. Thus, if a commit N is created over a commit $(N - 1)$, the existence of commit N in a given repository implies that commit $(N - 1)$ also exists in the repository. With this said, by checking the existence of commits in the local repository not yet replicated to the remote repository, and vice-versa, it is possible to come up with one of the situations presented in Table 2.

To illustrate how this approach works, let us assume that each commit is represented by an integer number and take the right portion of Figure 1, which represents developers led by Wolverine. This scenario is shown in Figure 37.

Table 1 – Possible states of a repository











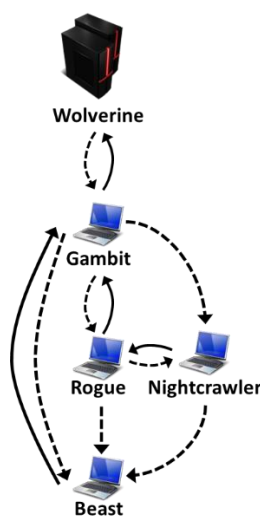
Status	Description
	DyeVC has not analyzed the repository yet.
	Repository is synchronized with all peers.
	Repository has changes that were not sent yet to its peers (it is ahead its peers).
	Peers have changes that were not sent yet to the repository (it is behind its peers).
	Repository is both ahead and behind its peers.
	Invalid repository. This happens when DyeVC cannot access the repository. The reason is presented to the user.

Table 2 – Status of a local repository regarding a remote one, based on the existence of non-replicated commits

Existence of non-replicated commits		Local Status
Local Repository	Remote Repository	
Yes	Yes	 Ahead and Behind (needs <i>pull</i> and <i>push</i>)
Yes	No	 Ahead (needs <i>push</i>)
No	Yes	 Behind (needs <i>pull</i>)
No	No	 Synchronized

**Figure 37 – Developers led by Wolverine**

At a giving moment, the local repositories of each developer have the commits shown in Table 3.

Table 3 – Existing commits in each repository

Repository	Wolverine	Gambit	Rogue	Nightcrawler	Beast
Commits	10 11	10 11	10 12	10 11 13	10

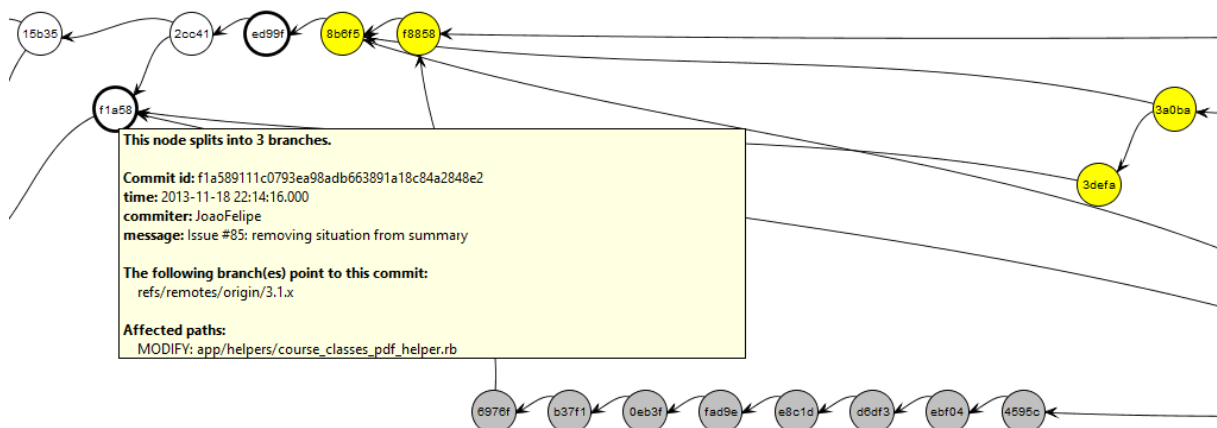
Considering just the synchronization paths presented in Figure 37, the perception of each developer regarding to their known peers is shown in Table 4. Notice that the perceptions are not symmetric. For instance, as Gambit does not pull updates from Nightcrawler, there is no sense in giving him information regarding Nightcrawler.

Table 4 – Status of each repository based on known remote repositories

Repository	Wolverine	Gambit	Rogue	Nightcrawler	Beast
Wolverine	-	-	-	-	-
Gambit	✓	-	-	-	-
Rogue	-	↕	-	-	-
Nightcrawler	-	✓	↕	-	-
Beast	-	↓	↓	↓	-

3.3.4 LEVEL 4: COMMITS

Level 4 complements information of Level 3 in order to provide an answer to Question Q3, by presenting a visual history of the repository (Figure 38) as a directed acyclic graph (DAG). Each vertex in the graph represents a known commit for the same project, which is named after its hash's five initial characters. A thicker border denotes that the commit is a branch's head (e.g., commit f1a48, for which the balloon is showing additional information).

**Figure 38 – Commit history for a given project**

Commits are drawn according to their precedence order. Thus, if a commit N is created after a commit $(N - 1)$, then commit N will be located in the right hand side of commit $(N - 1)$. For each commit, DyeVC presents the information shown in Figure 33 (gathered from the central database), along with information that is read in real time from the repository metadata, such as branches that point to that commit and files that were affected (modified, deleted, or inserted).

It is important to notice that this visualization contains all commits of all clones in an integrated graph, and not only commits that exist locally at the user's clone. Each commit is painted according to its existence in the local repository and in the peers' repositories. Ordinary commits that exist locally and in all peers are painted in white. Green commits are ready to be pushed, as they exist locally but do not exist in any peer in the push list. Yellow commits need attention because they exist in at least one peer in the pull list, but do not exist locally, meaning that they may be pulled. Red commits do not exist locally and are not available to be pulled, as they exist only in repositories that are not peers. Finally, gray commits exist locally (either on the user's computer or on a partner's computer), but belong to non-tracked branches, meaning that they can neither be pushed nor pulled.

This visualization can easily have thousands of nodes, one for each commit in the topology. Nevertheless, despite the high amount of nodes, users are generally interested in the most recent commits. As we show the commits following a chronological order, from left to right, most recent commits will be at the right part of the visualization, and DyeVC positions the graph so that these commits are shown when opening the visualization. More details regarding how this visualization is built can be found in Appendix A.

There is also the possibility to collapse nodes manually to provide a better understanding of huge amounts of data. As shown in Figure 39, the label of collapsed nodes show the number of contained nodes (there is a white node containing 118 commits and a green node containing 24 commits).

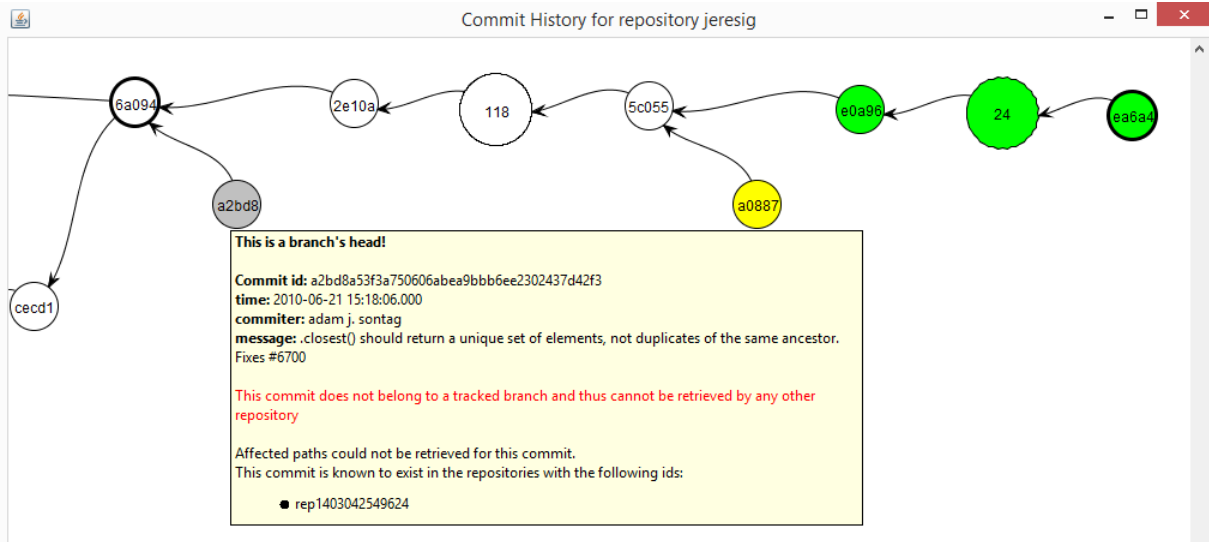


Figure 39 – Collapsed commit history

3.4 HOW INFORMATION IS GATHERED

Algorithm 1 shows the algorithm to update commits in the topology. This update finds out the existing commits and depicts where they can be found. The process was designed aiming at minimizing the amount of data transferred to the central database through the network. To accomplish that, we used Set Theory to check which commits have to be inserted, updated, or deleted in the central database. This process is executed for each repository clone *rep* being monitored by DyeVC. The algorithm receives the repository clone being monitored (*rep*), the set of existing commits in the database (*db.commits*), the set of existing commits at *rep* at the previous monitoring cycle (*previousSnapshot*), and the set of existing commits at *rep* at the current monitoring cycle (*currentSnapshot*).

First of all, we subtract *currentSnapshot* from *previousSnapshot* to find *commitsToDelete*, which contains commits that were deleted since the previous monitoring cycle (line 2), and we delete them from the database in order to cover the rare situations where a commit is deleted (line 3). Conversely, we subtract *previousSnapshot* from *currentSnapshot* to find *newCommits*, which contains commits that are new in *rep* since the previous monitoring cycle (line 5).

Algorithm 1: Updating commits in the topology

Input: a `RepositoryInfo` *rep* representing the repository clone being analyzed and three sets of `CommitInfo` *db.commits*, *previousSnapshot* and *currentSnapshot*.

```

1: begin
2: commitsToDelete  $\leftarrow$  previousSnapshot  $\setminus$  currentSnapshot
3: delete commitsToDelete from database
4:
5: newCommits  $\leftarrow$  currentSnapshot  $\setminus$  previousSnapshot
6: commitsToInsert  $\leftarrow$  newCommits  $\setminus$  db.commits
7: commitsToUpdate  $\leftarrow$   $\{c \mid c \in db.commits \wedge ((rep.pullsFrom \cup rep.pushesTo) \setminus c.foundIn) \neq \emptyset\}$ 
8:
9: for each c  $\in$  commitsToInsert do
10:   UPDATEFOUNDIN(c, rep, currentSnapshot)
11: end for
12:
13: for each c  $\in$  commitsToUpdate do
14:   UPDATEFOUNDIN(c, rep, currentSnapshot)
15: end for
16:
17: insert commitsToInsert into database
18: update commitsToUpdate in database
19: orphanedCommits  $\leftarrow$   $\{c \mid c \in db.commits \wedge c.foundIn = \emptyset\}$ 
20: delete orphanedCommits from database
21: end
22:
23: procedure UPDATEFOUNDIN(c: CommitInfo, rep: RepositoryInfo, currentSnapshot: Set of CommitInfo)
24:   isAhead  $\leftarrow$   $c \in currentSnapshot \wedge \exists r (r \in rep.pushesTo \wedge c \notin r.commits)$ 
25:   isBehind  $\leftarrow$   $c \notin currentSnapshot \wedge \exists r (r \in rep.pullsFrom \wedge c \in r.commits)$ 
26:
27:   if isBehind then
28:     c.foundIn  $\leftarrow$  c.foundIn  $\cup$   $\{r \mid r \in rep.pullsFrom \wedge c \in r.commits\}$ 
29:   end if
30:   if isAhead then
31:     c.foundIn  $\leftarrow$  c.foundIn  $\cup$   $rep \cup \{r \mid r \in rep.pushesTo \wedge c \in r.commits\}$ 
32:   end if
33:   if  $(\neg(isBehind) \wedge \neg(isAhead))$  then
34:     if  $c \notin currentSnapshot$  then
35:       c.foundIn  $\leftarrow$   $((c.foundIn \setminus rep) \setminus rep.pushesTo) \setminus rep.pullsFrom$ 
36:     else
37:       isTracked  $\leftarrow$   $\exists b (b \in rep.branches \wedge c \in b.reachableCommits)$ 
38:       if isTracked then
39:         c.foundIn  $\leftarrow$  c.foundIn  $\cup$   $rep \cup rep.pushesTo \cup rep.pullsFrom$ 
40:       else
41:         c.foundIn  $\leftarrow$  rep
42:       end if
43:     end if
44:   end if
45: end procedure

```

Next, we find out which of *newCommits* will have to be inserted into the database, by subtracting the existing commits in the database (*db.commits*) from *newCommits* (line 6). This step is necessary because some of the new commits might have already been inserted into the database by another instance of DyeVC running elsewhere. Commits that might be updated are represented by *commitsToUpdate* (line 7) and they consist of those commits that exist in the database, but were not found in at least one of the repository clones related to *rep* on the last monitoring cycle. These commits must be verified because since the previous monitoring cycle it may happen that they now are found in other repository clones related to *rep*.

Commits to be inserted or updated must be verified to check where they exist, thus updating the *c.foundIn* attribute. This verification is done using the procedure *updateFoundIn* (lines 23-45), which is called in lines 9-15. This procedure finds out where each commit *c* exists based on its existence locally or in any repository clone in the push or pull sets. This procedure verifies if *rep* is ahead of any clone in its push list regarding *c* (line 24), i.e., if *c* exists and if there is at least one clone that *rep* pushes to that does not contain *c*. Likewise, it verifies if *rep* is behind of any clone in its pull list regarding *c* (line 25), i.e., if *c* does not exist locally and if there is at least one clone that *rep* pulls from that contains *c*. If *rep* is behind, then all clones in *rep*'s pull list that contain *c* are added to *c.foundIn* (lines 27-29). If *rep* is ahead, then *rep* and all clones in *rep*'s push list that contain *c* are added to *c.foundIn* list (lines 30-32). It may happen that *rep* is neither ahead nor behind any clone (line 33). In such case, one of the following three scenarios may happen: In scenario 1, *c* does not exist in current snapshot (line 34), meaning that it also does not exist in any of the related clones, thus we remove *rep* and all its related clones from *c.foundIn* (line 35). For scenarios 2 and 3, we first depict if *c* is reachable from a tracked branch, i.e., if at least one of *rep.branches* is tracked and has *c* as one of its elements (line 37). In scenario 2, *c* is in a tracked branch, meaning that it also exists in all related clones (remember that *rep* is neither ahead nor behind their partners), thus we include *rep* and all its related clones in *c.foundIn* (line 39). Finally, in scenario 3, *c* is not in a tracked branch, meaning that it exists only in *rep*, thus we include only *rep* in *c.foundIn* (line 41).

After updating where each commit is found, *commitsToInsert* are inserted into the database (line 17) and *commitsToUpdate* are updated in the database (line 18). Finally, it may happen that some commits end up with an empty *foundIn* attribute, meaning that they do not exist anywhere in the topology (line 19). These so-called *orphanedCommits* are then removed from the database (line 20) and the algorithm ends.

At this point, it is worth mentioning two operations that do not occur very often while working with DVCSs, but exist and DyeVC supports them: *rebase* and *cherry picking*. A rebase

can be used to move a branch to a new base commit, or to clean up the repository history by transforming a series of commits into a single one. In both cases, what happens in fact is that all commits involved in the rebase operation are deleted, and new commits are created, with different hashes. It is strongly recommended that this kind of operation be executed before pushing the involved commits, because it demands replacing the old commits with new ones and it would look like part of the repository history abruptly vanished (CHACON, 2009). Since the commits involved in the rebase operation are deleted, DyeVC will detect this situation by comparing the current repository snapshot with the previous one and making the appropriate removals and insertions in the database.

Cherry picking is an operation that takes a commit from somewhere else, and “play it back” after the current branch’s head. This operation introduces the same changes made at the original commit, but with a different parent, which generates a new commit, with a different hash. Thus, as a new commit is generated, and DyeVC is able to register this new commit in the database.

3.5 IMPLEMENTATION DETAILS

We implemented our approach as a Java Swing application (ELLIOTT *et al.*, 2002) launched by Java Web Start Technology (MARINILLI, 2001). It currently monitors Git repositories, as it is the most used DVCS nowadays (ECLIPSE FOUNDATION, 2013). The source code and the link to download the tool via Java Web Start can be found at GitHub¹⁹. Appendix B presents basic usage information about the main features of DyeVC. The application gathers information from repositories using JGit library²⁰, which allows the user to use our approach without having a Git client installed. Information gathered is stored in a central document database running MongoDB (CHODOROW, 2013).

Our current deploy of DyeVC uses a MongoDB instance hosted at MongoLab²¹. To prevent from firewall blocks when accessing the database, we did not use MongoDB proprietary API, which would demand opening specific ports to connect to MongoDB. Instead, we chose to use MongoLab’s RESTful API. RESTful APIs (FIELDING, 2000) have the advantage to be available using standard HTTP and HTTPS protocols. This way, our approach can be used from inside corporate and academic environments, without major problems. In order to use the

¹⁹ <https://github.com/gems-uff/dyevc>

²⁰ <http://www.eclipse.org/jgit>

²¹ <http://mongolab.com>

RESTful API provided by MongoLab, we implemented a `MongoLabProvider`, which translates the application methods into RESTful commands and vice-versa. This provider also serializes and deserializes the application objects to and from JSON²² representations in order to send and receive them through the RESTful commands.

We present the information gathered as a series of graphs by using JUNG (*Java Universal Network/Graph Framework*) library²³, from which it inherits the ability to extend existing layouts and filters. All graphs present similar behavior, allowing the window to be zoomed in or out, whether the user wants to see details of a particular area or an overview of the entire graph. By changing the window mode from *transforming* to *picking*, it is possible to select a group of nodes and collapse them into one node, or simply drag them into new positions to have a better understanding of an area where there are too many crossing lines.

3.6 FINAL CONSIDERATIONS

In projects that use DVCS, there may be several clones where changes are being inserted simultaneously. These clones may communicate with each other indistinctively, turning the administration of such environment into a tough task. Today, administrators have no way to visualize the various clones and their dependencies, and developers have limited choices to provide awareness regarding parallel changes.

In this chapter, we presented the DyeVC approach, which supports the development and administration under DVCS environments, providing awareness in a non-obtrusive way, enabling administrators to monitor and visualize the repository topology and establishing a platform to present information and metrics. We also discussed aspects of its implementation and usage. DyeVC is currently able to monitor Git repositories, but it can be extended to support other DVCSs by using generic interfaces.

Table 5 compares DyeVC features with each category used to classify related work presented in Section 2.4. The following features were used in the comparison:

- **Notifications:** What does the approach notify?
- **CVCS:** Does the approach support CVCS?
- **DVCS:** Does the approach support DVCS?

²² <http://json.org>

²³ <http://jung.sourceforge.net>

- **Related repositories:** Does the approach identifies related repositories?
- **Levels of detail:** Does the approach present information in different levels of detail?
- **Multiple peers:** Does the approach support repositories with multiple peers (multiple pull / push destinations)?
- **Commits in peer nodes:** Does the approach detects commits in peer nodes (nodes that have a direct communication path among each other)?
- **Commits in non-peer nodes:** Does the approach detects commits in non-peer nodes (nodes that do not have a direct communication path among each other)?
- **Multiple branches:** Does the approach support multiple branches in DVCS?
- **Topology:** Does the approach supply any topology visualization that shows dependencies among repositories?
- **Commit History:** Does the approach allow visualizing only a partial commit history, showing only local commits, or does it allow visualizing a full commit history, including commits in other repositories that were not synchronized yet, or that are in non-tracked branches?

Table 5 – Comparing DyeVC features with related work

Feature / Category	Commit Notification	Awareness of Concurrent Changes	Repository Visualization	DVCS clients	DyeVC
Notifications	new commits	Conflicts	No	No	Status change against peers
CVCS	Yes	Yes	Yes	No	No
DVCS	Some ²⁴	Some ²⁵	Some ²⁶	Yes	Yes
Related repositories	No	No	No	No	Yes
Levels of detail	No	No	No	No	Yes

²⁴ Exceptions are SCM Notifier and Hg Commit Monitor

²⁵ Exception is Crystal

²⁶ Exceptions are VisGi, Visugit and GitHub's Network Graph

Feature / Category	Commit Notification	Awareness of Concurrent Changes	Repository Visualization	DVCS clients	DyeVC
Multiple peers	No	No	No	No	Yes
Commits in peer nodes	No	Some ²⁷	Some ²⁸	No	Yes
Commits in non-peer nodes	No	No	No	No	Yes
Multiple branches	No	No	No	Yes	Yes
Topology	No	No	No	No	Yes
Commit history	No	No	Some ²⁹ / Partial ³⁰	Partial ³⁰	Full

At first sight, it may seem that DyeVC is very similar to *Crystal*, one of the related works discussed in Section 2.4.2, especially due to the icons used in the visual interface to indicate status (compare Figure 19 on page 34 with Figure 36 on page 49, and with Table 1 on page 50). Both approaches work with DVCSs (besides Git, *Crystal* also supports Mercurial) and use working copies to perform analysis without affecting the user's repositories. Apart from these similarities, there are major differences between them, as follows:

- **Goal:** *Crystal's* goal is to identify conflicts among pairs of repositories (local repository versus central repository or peer's repository), whereas *DyeVC's* goal is to provide awareness regarding the existing peers and their synchronization, in different levels.
- **Repository identification:** *Crystal* demands the user to point out every repository they want to compare. For instance, when working with four repositories, all of them need to be manually registered in the approach. *DyeVC* does not require every repository to be registered, because it automatically looks at configuration files to

²⁷ Exception is Lighthouse

²⁸ Exception is GitHub's Network Graph

²⁹ Visugit and GitHub's Network Graph

³⁰ Approaches allow visualizing only local commits. Commits in other repositories that were not synchronized yet, or that are in non-tracked branches, are not shown.

discover all the repositories that one pushes to or pulls from. *DyeVC* works even with scenarios where a repository pushes to or pulls from more than one partner.

- **Repository comparison:** *Crystal* analyses one repository against several peers, on a pair basis, whereas *DyeVC* analyzes all repositories against each other, provided that there are pushes or pulls among them.
- **Information type:** *Crystal* provides information regarding the synchronization between two repositories and the existence of conflicts. *DyeVC* does not report conflicts natively, but it shows synchronization information in different levels of detail (repositories, branches, and commits – see more at Section 3.3), including a topology view, which shows all known repositories and their interdependencies.
- **Multiple branches:** *Crystal* deals with only one branch at a time. If one wants to make comparisons involving different branches, it would be necessary to create different working copies of the same repository, each one pointing to a different branch, and register these working copies so that the approach can monitor them. *DyeVC* automatically checks all tracked branches in all registered repositories.
- **Allowed actions:** While *Crystal* provides commands to *push*, *pull*, *compile*, and *test* a repository, *DyeVC* allows one to visualize branches status, topology, and history.

CHAPTER 4 – EVALUATION

4.1 INTRODUCTION

We conducted two experiments and an observational study to evaluate DyeVC. Besides using the evaluation results to identify limitations and improvements in our approach, we aim at verifying if DyeVC is capable of answering the questions posed in Section 1.2:

- Q1: Which clones were created from a repository?
- Q2: What are the dependencies between different clones?
- Q3: Which changes are under work in parallel (in different clones or different branches) and which of them are available to be incorporated into my work?

We also aim at answering the question related to non-function aspects of our approach, which was also introduced in Section 1.2:

- Q4: Is it computationally feasible to gather this information from all known repositories, keeping them available to be used when needed?

In the first experiment, we conducted a *post hoc* analysis over the JQuery³¹ open-source project, to check if DyeVC can help answering questions Q1-Q3. Next, we conducted an observational study involving four students that used DyeVC. This study also used the JQuery project. Finally, we took some open-source projects of different sizes and from different sources, in order to evaluate the scalability of our approach, aiming at answering question Q4.

This chapter is organized as follows: Section 4.2 describes the *post hoc* analysis. Section 4.3 presents the observational study. Section 4.4 describes the performance evaluation of our approach. Section 4.5 discusses some threats to validity of the experiment. Lastly, Section 4.6 presents the final considerations of this chapter.

4.2 ANALYZING *JQUERY* PROJECT WITH DYEVC

We conducted a *post hoc* analysis using an open source project to demonstrate that our approach can help answering questions Q1-Q3. We used the JQuery project, a project that began in 2006 and had 6,222 commits by the time of the evaluation. We reconstructed the repository

³¹ <https://github.com/jquery/jquery.git>

history, simulating the actions that occurred in the past. We do not replicate the repository history here, due to its size, but the repository is public available at Github. Automatically generated comments helped us to depict specific flows that occurred in the past. For example, the comment “Merge branch 'master' of <https://github.com/scottjehl/jquery> into *scottjehl-master*” tells us that there was a user named *scottjehl* and that the merge operation was done at a branch called *scottjehl-master*. Although one might perform a merge manually and insert a different text in the comment, this did not compromise our analysis because we had a focus on depicting some of the merge situations, and not all of them.

Due to the operating mode of Git, some details are missing, but these details do not compromise our analysis. The first one is the moment when a clone arises or decesses. This information does not exist anywhere in the repository. We inferred the creation of clones looking at the commit messages in the repository history (a commit by developer X led to the creation of a clone named X). Clones created at a given time stayed alive for the rest of the analysis.

The second missing detail is that, although we had the commit dates and times in the repository history, these dates and times were not guaranteed to be correct. This occurs because DVCSs do not have a central clock. Each commit is registered with the local time at the machine where the clone is located, which could lead to commits in the history with a predecessor in the future, depending on when and where each one of them were performed. This missing detail is not important, because the precedence between two commits is not depicted from their commit times, but from the pointers that Git maintains from a commit to its parents. We can use these dates, but not as an authoritative information.

We chose a moment in time when three developers were involved, performing commits and merging changes in the repository. We created three clones for these developers, named after their author names and commit messages: *jesesig*, *adam*, and *aakoch*. Figure 40 shows the topology view on Sep 24 2010, when *aakoch* had 121 commits pending to be pushed to the central repository (hereafter represented as *central-repo*). Figure 41 shows part of *aakoch*'s commit history and how DyeVC represents commits pending to be pushed as green nodes in the graph.

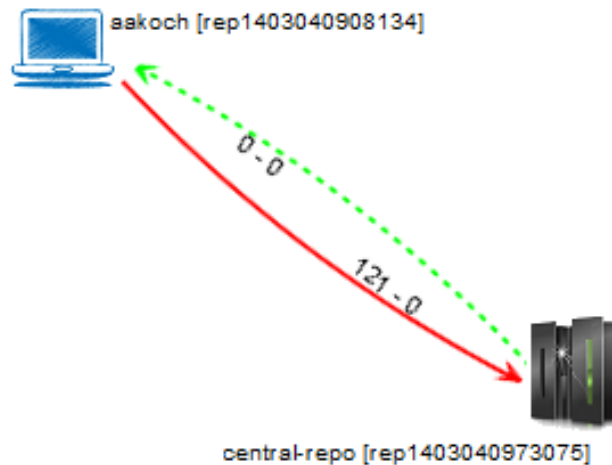


Figure 40 – Topology view showing first monitored repository (Sep 24 2010)

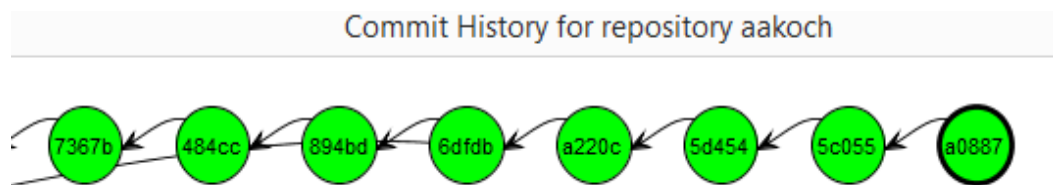


Figure 41 – aakoch's commit history showing commits pending to be pushed

Later on, *aakoch* pushed his commits to *central-repo*. Adam performed a commit on Jun 21 2010 (before *aakoch*'s push to *central-repo*), but he did not push this commit to *central-repo*. On Sep 27 2010, *jeresig* committed some changes as well (after *aakoch*'s push), and did not push them either. At this moment, we registered them to be monitored by DyeVC. Figure 42 shows the topology view after this registration on Sep 27 2010. Here, we can see that *aakoch* was synchronized with *central-repo*, whereas *adam* and *jeresig* had some pending actions.

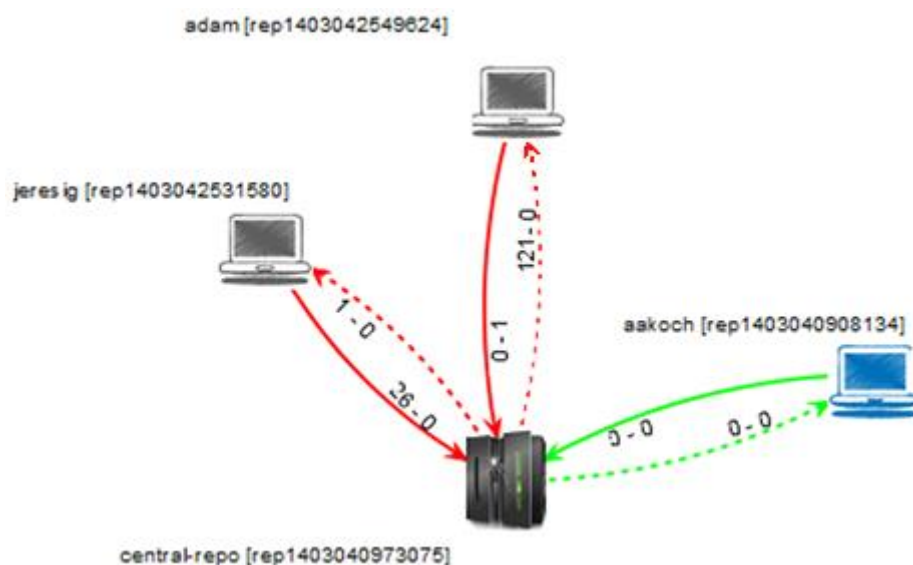


Figure 42 – Topology view showing the three monitored repositories (Sep 27 2010)

At this point, we are able to revisit questions Q1 and Q2:

Q1: Which clones were created from a repository? DyeVC's topology view (Figure 42) shows all the clones where there is an instance running, and discovers other clones connected to them, even if there is no instance running.

Q2: What are the dependencies between different clones? DyeVC's topology view (Figure 42) shows the dependencies between the peers in the topology, as well as the number of commits ahead or behind in each of these dependencies.

Adam had 121 commits to pull from *central-repo*, what is corroborated by the details of his tracked branches (master branch in Figure 43). He also had a non-tracked commit pending to be pushed. Non-tracked commits are not shown in the tracked branches view, but we can see them in commit history views, painted in gray. Figure 44 shows the collapsed commit history for *jeresig*, where we can see *adam*'s non-tracked commit with hash *a2bd8*. We can see in the details box showing the commit details that the committer was *adam j. sontag*. The details also show the following message in red letters: "*This commit does not belong to a tracked branch and thus cannot be retrieved by any other repository*". The repository where this commit exists is shown at the end of the details box (*repl403042549624*). We know this is *adam*'s repository by comparing this id with *adam*'s repository id in Figure 42).

The repository history leads us to think that *jeresig* is a core developer in the project, because he performed most of the merges to master branch. Looking at Figure 42, we see that he had 26 commits pending to be pushed to *central-repo*. These 26 commits can be seen at *aakoch*'s commit history (Figure 45), as red commits, once they could not be pulled by *aakoch* until *jeresig* had pushed them to *central-repo*. There was also a commit in *central-repo* pending to be pulled by *jeresig*. If we look at Figure 44 we see that the only yellow commit is *a0887*, made by *aakoch*. This tells us that *jeresig* pulled changes from *central-repo* at a moment before *aakoch* pushed commit *a0887*. This analysis make us return to the discussion we had after Figure 42. There, our conclusion was that *aakoch* had pushed all pending commits at once, and then *jeresig* pulled these commits. However, if this was the case, *jeresig* would already have commit *a0887*, but the last commit from *aakoch* that *jeresig* has is *5c055* (the white node just before commit *a0887* in Figure 44). Thus, we conclude that what happened in fact was that *aakoch* pushed all commits up to commit *5c055*, *jeresig* pulled these commits and, later on, *aakoch* performed commit *a0887* and pushed it, leaving *jeresig* with no awareness of this action.

DyeVC

File View Help

Monitored repositories

Status	System Name	Clone Name	ID	Clone Path
✓	jquery	aakoch	rep1403040908134	F:\evaluation\repos\aaakoch
↕	jquery	jeresig	rep1403042531580	F:\evaluation\repos\jeresig
↓	jquery	adam	rep1403042549624	F:\evaluation\repos\adam

ID: rep1403042549624
 Address: F:\evaluation\repos\adam
 Last checked on: 2014-06-17 19:21:29

↓ Behind: 121
 Local branch: master Remote branch: refs/remotes/origin/master
 Remote url: F:/evaluation/repos/central-repo

Figure 43 – Adam’s tracked branches

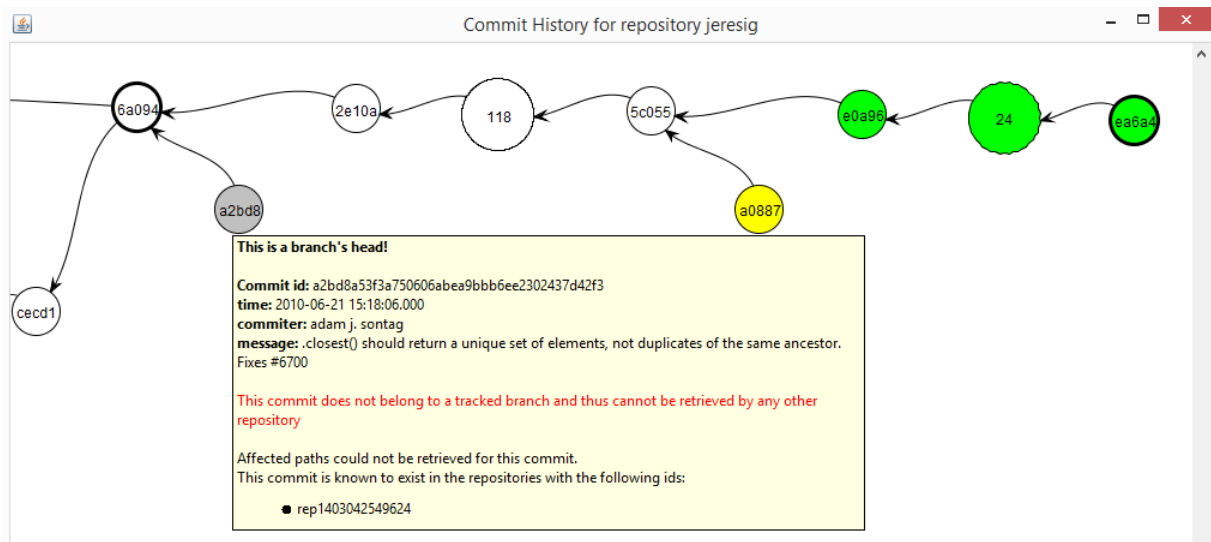


Figure 44 – Jeresig’s collapsed commit history

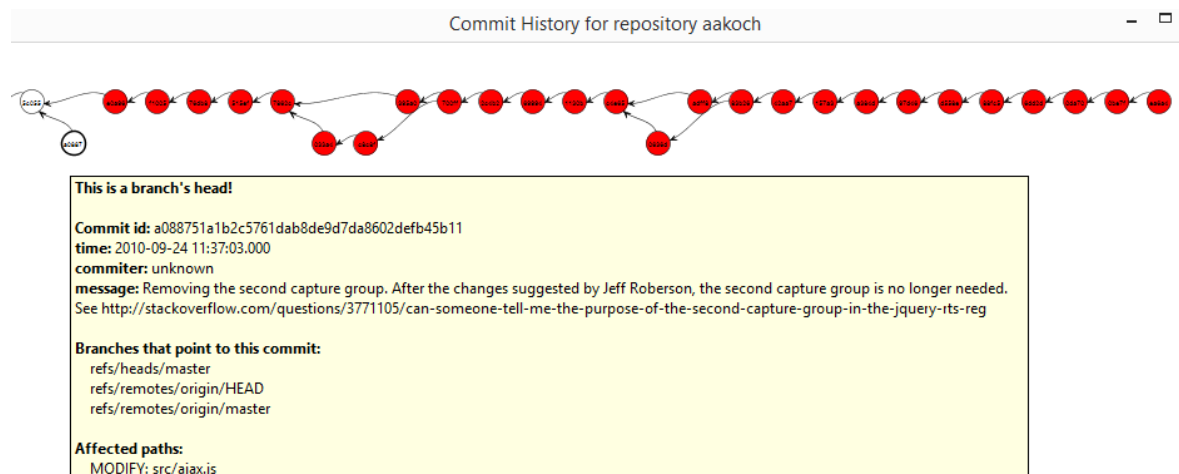
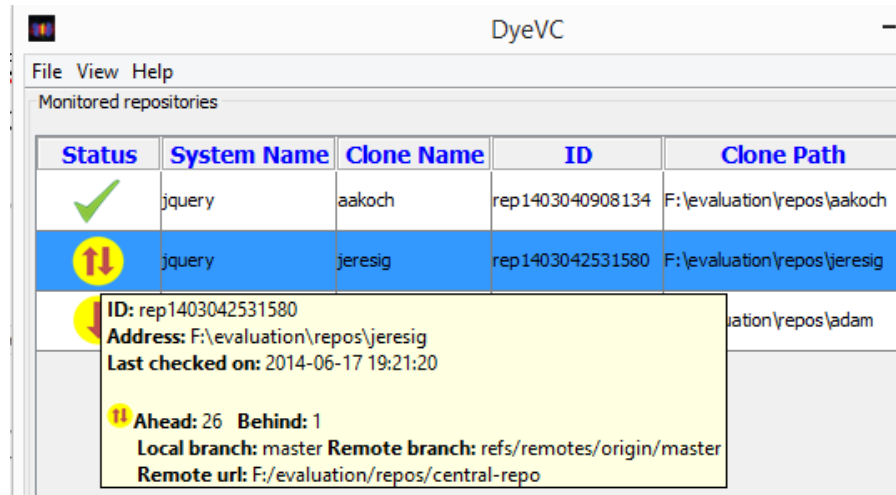


Figure 45 – Aakoch’s commit history

If we look at Figure 46, we see that all pending commits (those that were pending to be pushed and those that were pending to be pulled) are related to the same branch (*master*). This tells us that, if *jeresig* wanted to push these commits to *central-repo*, he would have to perform both push and pull operations. This analysis helps us revisit and answer Q3.



Status	System Name	Clone Name	ID	Clone Path
✓	jquery	aakoch	rep1403040908134	F:\evaluation\repos\aaakoch
⬆⬆	jquery	jeresig	rep1403042531580	F:\evaluation\repos\jeresig
ID: rep1403042531580 Address: F:\evaluation\repos\jeresig Last checked on: 2014-06-17 19:21:20 ⬆⬆ Ahead: 26 Behind: 1 Local branch: master Remote branch: refs/remotes/origin/master Remote url: F:/evaluation/repos/central-repo				

Figure 46 – Jeresig’s tracked branches

Q3: Which changes are under work in parallel (in different clones or different branches) and which of them are available to be incorporated into my work? New commits in tracked branches of peers can easily be found looking at Level 3 information (tracked branches, shown in Figure 43 and Figure 46). This view shows to which branch these commits are related and how many new commits exist. If we want to look at each commit individually, we can look at Level 4 information (commit history, shown in Figure 41 and Figure 45) and notice the yellow nodes. Additionally, Level 4 information is used to find new commits in repositories that are not peers (red nodes), or new commits in non-tracked branches (gray nodes).

4.3 OBSERVATIONAL STUDY

We conducted an observational study over the same project used in the *post hoc* analysis (JQuery) to analyze the capability of the visualizations provided by DyeVC in supporting developers and repository administrators. In this kind of study, the subject performs tasks while an experimenter observes the actions taken. This kind of study aims at collecting information regarding how a given task is performed (SHULL *et al.*, 2001). This information can help in understand how a new process is used.

This Section is organized as follows: Section 4.3.1 describes the study. Section 4.3.2 presents the procedure used during the study. Section 4.3.3 presents the study results. Finally, Section 4.3.4 presents the subjects evaluation regarding the study and the approach.

4.3.1 DESCRIPTION

The pre-requisite to participate in this study was to have experience in any DVCS. The study was conducted with four volunteers. All of them are graduate students from the Software Engineering area at Universidade Federal Fluminense (UFF). Although there were no industry subjects involved, (SVAHNBERG *et al.*, 2008) discuss that students may work well as subjects in empirical studies. There was no compensation of any kind to the subjects. Four different sessions were conducted, each of them with one subject. All forms presented in this study are written in Portuguese, which was the native language of all subjects.

The goal of this observational study was to analyze when DyeVC helps on understanding the project history better than existing tools. This study was divided in two phases, each one with two scenarios, where the subject had to answer a number of questions related to usual work with DVCS. The scenarios in each phase were the same and in each one of them the subject played a different role. In Scenario One, the subject played the developer role, working in a clone named *aakoch*. In Scenario Two, the subject played the repository administrator role.

In Phase 1, the subject had a number of questions to be answered about the JQuery project history, before knowing DyeVC. The subject could use any desired DVCS client to answer these questions among the ones available in the test machine: *gitk*, *Tortoise Git*, *Git Bash*, and *SourceTree*. It was also possible to access the Internet to search for any other procedure or tool that could help in answering the questions. In Phase 2, DyeVC approach was presented and the subject used it to help answering the same questions. The possible answers in Phase 2 were either “keep the answer of Phase 1”, meaning that using DyeVC did not change the subject perception, or a different answer, meaning that using DyeVC actually changed the subject perception.

The main points to be observed in this study are:

1. What procedure subjects followed to answer the questions in Phase 1;
2. Which (if any) of the questions in Phase 1 could not be answered;
3. If DyeVC let subjects answer any questions in Phase 2 that were not previously answered in Phase 1;
4. The overall evaluation that DyeVC received from subjects.

4.3.2 PROCEDURE

Initially, the subject filled the Informed Consent Form (Appendix C) and, upon accepting the terms, they filled the Characterization Form (Appendix D). Next, the subject received the document entitled Activities – Phase 1 (Appendix E) and was allowed to read the first two sections of the document (Instructions and Context). Phase 1 of this study included answering the questions for both scenarios.

After finishing Phase 1, the subject watched a 10-minute video presenting DyeVC. Next, the subject received the document entitled Activities – Phase 2 (Appendix F). This document contained a summarization of DyeVC approach, its main visualizations, and the meaning of each different icon, node, edge, etc. In Phase 2 of this study, the subject used DyeVC and contrasted the answers given in Phase 1 with the answers found using DyeVC. The set of questions in Phase 2 was the same as in Phase 1. The subject could either answer that they want to keep the previous or write a new answer, based on the analysis done using DyeVC.

Finally, the subject filled the Exit Survey (Appendix G), expressing their opinion regarding the tasks executed and the DyeVC approach. Each subject spent one hour on average in this study.

4.3.3 RESULTS

Four graduate students participated in this study and all of them had experience in software development and DVCS. Table 6 summarizes the subjects' profile.

Table 6 – Summary of the Characterization Form

Criteria			P1	P2	P3	P4
1		Grade	D.Sc. in progress	M.Sc. completed	M.Sc. in progress	D.Sc. in progress
2	2.1	Type of projects in which spend most time	Academic	Academic	Academic	Academic
	2.2	Experience (Years)	+10	+10	+10	6-10
	2.3	Most used Version Control System	Git	Mercurial	Git	Git
	2.4	Average team size	Alone	6-10	2-5	Alone
3	3.1	How frequently changes are committed	After completing a feature	After completing a feature	After completing a feature	After completing changes in a class/method
	3.3	How changes are grouped or split into commits	By tasks	I have no opinion	By tasks	By classes

Table 7 presents time spent for each subject to answer each question of both scenarios from phases 1 and 2. Values include time to understand the question, investigate repositories with available tools, look for the answer and write down the answers in the form. Values do not include time spent filling the consent form and the characterization form, watching the video about DyeVC, and filling the Exit Survey.

Table 7 – Time spent to answer each question in the study

Subject	Time spent (in minutes)			
	Scenario 1		Scenario 2	
	Phase 1	Phase 2	Phase 1	Phase 2
P1	14	5	-	6
P2	13	6	-	5
P3	3	2	-	4
P4	10	2	-	10

It is possible to notice, by looking at Table 7, that all subjects took less time to complete Scenario 1 in Phase 2 (using DyeVC). For Scenario 2, times for Phase 1 are not shown because none of the subjects managed to answer the questions without using DyeVC.

Table 8 presents the expected answers to each of the question proposed in both phases. Questions 1.1 through 1.3 correspond to Scenario 1, whereas questions 2.1 through 2.4 are related to Scenario 2.

In Phase 1, each subject used different ways to look for the answers, which are detailed in next sections. In Phase 2, subjects correctly used DyeVC to find the answers. Question 1.1 was answered using DyeVC Level 3 visualization (Tracked branches). Question 1.3 was answered using Level 4 visualization (Commit History). Finally, questions 1.2 and 2.1 through 2.4 were answered using Level 2 visualization (Topology). Almost all subjects answered all the questions similarly, except for subject P4 in question 1.2 from Phase 1.

Table 8 – Expected answers to questions proposed in both phases

Question	Answer	
	Without DyeVC (Phase 1)	With DyeVC (Phase 2)
1.1 – What is the status of your clone, compared to the central repository?	121 ahead	Same as Phase 1
1.2 – Who else is working in JQuery project (other clones)?	I don't know	Nobody
1.3 – Which files were modified in commit with hash beginning in 5d454?	src/effects.js	Same as Phase 1
2.1 – What are the existing clones for JQuery project?	I don't know	Aakoch, Adam and Jeresig
2.2 – Which clones are synchronized with the central repository?	I don't know	Aakoch
2.3 – How many commits in tracked branches are pending to be sent to the central repository?	I don't know	26
2.4 – Is there any commit in non-tracked branches? Where?	I don't know	Yes (Adam)

Subject P1 answered questions 1.1 and 1.3 in Phase 1 using the command line interface. To answer question 1.1, they looked at the log for both local and remote repositories, counted down how many hashes there were in each log and subtracted these numbers to find the answer. Question 1.3 was answered with *git show* command, which shows, for each affected file in the commit, what has changed. The answer to this question was easy to find because only one file was affected, but if many files had been affected, the subject would have trouble finding all affected files using this procedure. For questions 1.2 and 2.1 through 2.4, the subject tried to find a way to discover related clones by searching the Internet. After a few searches with no promising results, the subject gave up and their answer was “I don't know”. Once there was no answer to question 2.1, next questions in Scenario 2 could not be answered as well.

Subject P2 answered question 1.1 by issuing the *git status* command. To answer question 1.3, they used *Tortoise Git* and walked through the commit tree until finding the desired commit. For questions 1.2 and 2.1 through 2.4, the subject answered that they didn't know a way to find an answer. When answering question 2.1, the subject commented that, as a repository manager, they should know which were the existing clones and their relationships, but they did not have any resources available to accomplish that.

Subject P3 answered question 1.1 by issuing a *git status* command (same as subject P2). To answer question 1.3, they used *Tortoise Git* but they found the desired commit using the search feature of the tool, instead of walking through the commit tree. For questions 1.2 and 2.1 through 2.4, the subject answered that it was not possible to find an answer.

Subject P4 answered questions 1.1 and 1.3 using *Sourcetree*. This subject answered question 1.2 differently from the others. They wrote down each different author of each commit as if it was a different clone. Although this is a valid interpretation, it may happen that authors commit changes in the same clone, and this would lead to a wrong answer for this question. For questions 2.1 through 2.4, the subject answered that it was not possible to find an answer.

The overall results of this study were positive, because subjects were able to answer correctly questions 1.1 and 1.3 whether using DyeVC or not. Also, further questions, which did not have a way to be answered without DyeVC, were answered correctly by using the approach.

By looking at the results and at the points to be observed (mentioned in Section 4.3.1), we can say that:

1. Each subject uses a different tool and follows a different procedure to find answers regarding DVCS usage;
2. In Phase 1, questions that depend on information of peers cannot be answered effectively without using DyeVC;
3. DyeVC let subjects answer all questions in Phase 2, including those that had not been answered before.

4.3.4 SUBJECTS EVALUATION

After finishing their tasks, all subjects were asked to fill an evaluation form. The overall evaluation was positive. All subjects found easy to interact with DyeVC, to identify related repositories, and to use the operations the approach provides. They also stated that DyeVC visualizations were useful to answer the questions and that DyeVC helped investigating the *JQuery* project (one of the subjects answered that the help was neutral).

All subjects chose topology visualization as the most helpful visualization in DyeVC. This choice was made by two reasons: the ability to find out relationships between peers and the ability to know which repositories have pending pushes.

The following positive aspects of DyeVC were mentioned:

- “It helps visualizing peers and their relationships”;
- “It helps visualizing the status of each clone”;
- “It is easy to use”;
- “It is useful to manage projects”;

- “It provides a way to know that people are working in parallel”; and
- “It let us view non-tracked branches and commits in all the topology, regardless of clones where they can be found”.

The following negative aspects of DyeVC were mentioned:

- “It does not detect people that do not use DyeVC”;
- “It violates privacy when showing information regarding non-tracked branches”; and
- “Having a centralized repository can lead to information overload and it go against the DVCS usage philosophy”.

Detecting people that do not use DyeVC is in fact possible, with some restrictions. As we discussed in Section 3.2, if DyeVC monitors a given repository *rep*, it is able to find all peers that relate to *rep*, either by push or pull relationships, even if they are not being monitored. The privacy violation related to non-tracked branches is partial, because the information that DyeVC gathers do not include the content of any files, just their names (see Section 3.2 for more information). Finally, the need for a centralized repository is to fill the gap that emerged with DVCS usage: not having a central point with information from all the clones of a given DVCS repository.

Regarding the study, one of the subjects stated that questions in Scenario 2 were biased, leading to the answer: “I don’t know”. This was, in effect, a natural side-effect of Scenario 2, where we tried to show that many questions that repository administrators can have are difficult or even impossible to answer with existing tools.

Finally, the subjects mentioned some improvements that could be incorporated in the approach:

- “Provide a visualization showing all people involved in the project (commit authors)”;
- Improve help screens, which are now text based, by including DyeVC icons in the explanations”;
- “Provide a way in topology view to see all hosts where clones are stored”;
- “Provide a way to search commits in Commit History visualization”

4.4 PERFORMANCE EVALUATION

In order to evaluate the scalability of our approach, we measured the time spent to perform the most common DyeVC operations, by analyzing projects with repositories of different sizes and hosted in different Git servers. Table 9 shows the monitored projects, the server hosting the original repositories and the repository metrics – number of commits, disk usage, and number of files. For each one of the monitored projects, we created a local clone the original repository. These local clones were used to perform measurements. All measurements were taken in the same period of the day and from the same machine, a Core Duo CPU running at 2.53 GHz, with 4GB RAM running Windows 8.1 Professional 64 bits, connected to the internet at 35 Mbit/s.

Table 9 – Monitored projects and repository metrics taken during evaluation

Project	Hosting	Repository metrics		
		# commits	Size (MB)	# files
DyeVC	github.com	187	1.0	539
Sapos	github.com	702	7.0	685
jgit	eclipse.org	2,979	10.0	1,595
egit	eclipse.org	3,775	27.0	1,478
jquery	github.com	5,518	20.0	253
Tortoise Git	code.google.com	6,166	85.0	3,220
Gitextensions	github.com	6,417	448.0	1,549
drupal	drupal.org	23,922	84.4	9,290
Expresso Livre	gitorious.org	25,822	141.0	20,729
Git	github.com	35,260	98.0	2,656

We measured the main operations of our approach. Table 10 shows the time spent to perform foreground operations and Table 11 shows the time spent to perform background operations. Table 10 also presents the memory usage during the execution of the “Commit History” operation. The measured operations were:

- “Commit History”, an operation invoked when the user requests to see the commit history of a given system.
- “Topology”, invoked when the user wants to see the topology of clones of a given system.
- “Insert 1st”, invoked when the user includes the first clone of a given system to be monitored.
- “Insert 2nd”, invoked when the user includes a clone to be monitored in a system that already have registered repositories.

- “Check Branches”, invoked periodically to check all the monitored clones, searching for ahead or behind commits.
- “Update Topology”, invoked periodically to update the topology information in the central database. It updates the existing clones, their peers, and the existing commits, marking in which clones each commit is found.

Table 10 – Time spent to perform foreground operations

Project	Foreground operations		
	Commit History		Topology
	Time (s)	Memory Usage (MB)	Time (s)
DyeVC	3.5	15	2.7
Sapos	5.6	19	3.2
jgit	18.4	512	3.4
egit	21.3	559	3.7
jquery	65.0	1,121	4.1
Tortoise Git	68.0	492	4.2
Gitextensions	73.0	1,529	17.0
drupal			18.0
Expresso Livre			18.2
Git			19.4

Table 11 – Time taken to perform background operations

Project	Background operations times (s)			
	<i>Insert 1st</i>	<i>Insert 2nd</i>	<i>Check Branches</i>	<i>Update Topology</i>
DyeVC	12.4	16.1	1.7	4.4
Sapos	20.8	22.6	1.8	5.2
jgit	42.4	46.0	5.9	6.8
egit	49.6	46.6	4.2	7.3
jquery	40.0	37.4	1.4	9.4
Tortoise Git	39.0	36.0	1.6	9.6
Gitextensions	155.8	129.0	1.6	10.6
drupal	102.0	95.0	2.0	18.0
Expresso Livre	110.0	102.0	2.1	19.3
Git	196.0	158.6	3.4	40.0

It is possible to notice by looking at Table 10 that the “Commit History” operation has no values for the last three projects. This occurs because, as the number of commits increases, more memory is used to calculate the commit history graph. The current algorithm has an $O(x^2)$ space complexity (x being the number of commits). The increasing memory usage is due to two factors: First, in order to plot the commit graph, JUNG library requires the entire graph in memory. Second, the x position of nodes in the graph are calculated based on node ancestry, but the y position is calculated in order to minimize the number of lines crossing during merges and splits in the graph. In order to do so, we used the implementation of *Dijkstra’s algorithm* (1959) provided by JUNG library, for which memory usage scales exponentially with the

number of nodes. Our test machine was configured with a two GB maximum Java Heap Size, which let us execute the “Commit History” operation for repositories with up to 6,417 commits, although the other operations could be executed for all the projects under analysis. This is an aspect for future improvements.

According to Table 10 and Table 11, the slowest operations were “Insert 1st” and “Insert 2nd”, due to the amount of data sent over the Internet to update the database. The tasks performed by both operations are almost the same, which involve storing the clone information in the central database (`RepositoryInfo` data, shown in Figure 33, on page 46) and creating a working copy of the monitored clone in the user’s home folder, where DyeVC will perform all necessary fetches during its operation. The major difference between the two operations is related to creating / updating the commit existence in the database. In the “Insert 1st” operation, all commits must be included in the database, whereas in the “Insert 2nd”, existing commits are updated to reflect in which clones they exist and only new commits are inserted. The time spent for both operations is almost the same, because inserting or updating commits in the database takes about the same amount of time.

The only operation with no significant variation in response times was “Check Branches”. Amongst the foreground operations, the “Topology” operation had a significant increase in its response time, but with lower values than the “Commit History” operation. This is because the latter deals with much finer grain data than the former.

Table 12 shows, for each measured operation, the correlation between time spent and each repository metric, according to the *Pearson coefficient* (PEARSON, 1895). This correlation coefficient measures the linear correlation between two variables x and y and ranges from -1 to 1 . Values of 1 or -1 mean that a linear equation can describe the correlation between the two variables perfectly (either positive or negative, respectively). A value of 0 means that there is no linear correlation between them.

Table 12 – Pearson coefficient between time spent and repository metrics for measured operations

Operation	# commits	Size	# files
Commit History	0.95	0.62	0.41
Topology	0.86	0.61	0.59
Insert 1st	0.79	0.65	0.30
Insert 2nd	0.82	0.65	0.36
Check Branches	0.00	-0.28	-0.13
Update Topology	0.94	0.17	0.33

Looking at Table 12, we can see that the number of commits in the repositories is the metric with the Pearson coefficient nearest to 1 . Generally, operations took longer in

repositories that had more commits. Size also presents a high Pearson coefficient, and it was expected that size and number of commits would present similar behaviors, because, as we discussed before, commits do not delete data from VCSs. Even if an artifact is logically deleted in a commit, it continues to exist in previous versions of the repository. Even in this case, supposing a commit that consists of only logical deletes, some metadata will be stored in the repository, causing it to grow continuously in space.

After this analysis, we are able to revisit and answer Q4.

Q4: *Is it computationally feasible to gather this information from all known repositories, keeping them available to be used when needed?* Yes. The current version of DyeVC allows one to gather information from repositories with different sizes, using up to 2 GB of memory, in a reasonable time (the whole topology is shown in about 17 seconds). Besides that, branch checking is performed on repositories with tens of thousands of commits in a few seconds (3.4 seconds for the *Git* repository, which had more than 35,000 commits). Lastly, it is possible to visualize the commit history for repositories with up to 6,417 commits. Increasing the amount of memory available or optimizing the algorithm are ways for allowing the approach to process a higher number of commits in this visualization.

4.5 THREATS TO VALIDITY

While we have taken care to minimize threats to the validity of the experiment, some factors can influence the results. The usage of a *post hoc* analysis to evaluate a real project may not reflect the exact sequence of events that occurred, although the outcome did not change. For example, when we say that *aakoch*, at some moment, had 121 commits pending to be pushed to the central repository, these commits could have been pushed at once, or by a series of smaller pushes.

Moreover, only one project was selected to perform the *post hoc* analysis, what imposes limitations from a statistical standpoint. Furthermore, there is a risk regarding the instrumentation used to measure the response times during the performance evaluation. As we used a database stored over the Internet, the response times may have been negatively affected by connectivity issues and network instability. The usage of a 35Mbit/s home network also contributes to this network instability, because home networks have much lower service level agreements than corporate ones.

Next, we used an open source project to perform the *post hoc* analysis. However, the *modus operandi* of peers in this context may be different from that of peers in academic or corporate contexts. Besides that, it is not possible to represent all different situations of a real

project. We discussed the most common situations that occur when using DVCSs, but a more thorough verification is needed to evaluate the usefulness of our approach in other situations.

The selection of subjects in the observation study was done by asking for volunteers from students in the same research group of the experimenter. This was necessary due to time and people restrictions. Therefore, this group might not be representative and can be biased regarding the experimenter. Moreover, there were few subjects in this study. Thus, the results may have been influenced by the size and by specific characteristics of the group.

Finally, tasks involving DyeVC were performed right after presenting the approach, giving no time to subjects to assimilate the tool. Results may have been influenced by this lack of time to mature the necessary knowledge to use the approach efficiently.

4.6 FINAL CONSIDERATIONS

The evaluation of DyeVC aimed at identifying if the approach helps developers and administrators to work in projects that involve DVCS. We showed that DyeVC could provide awareness regarding who are the people that work together on the same project and how they interact and / or depend on each other to accomplish their work. The observation study showed that DyeVC could effectively help developers and repository administrators by saving time and providing ways to help questions regarding DVCS usage that could not be answered before. We also showed that it is feasible to gather information from different repositories, consolidating and showing it at a reasonable time.

CHAPTER 5 – CONCLUSION

5.1 CONTRIBUTIONS

This work introduced a novel approach for DVCS monitoring and awareness, entitled DyeVC. This approach gathers information from registered DVCS clones and their peers, regarding the flow of communication and the existing commits in every node, and records this information in a central database.

The gathered information is consolidated, allowing developers to increase their knowledge of what is going on that might affect their work, as well as which changes have to be sent/received to/from their teammates. It also gives repository administrators the knowledge about which are the existing clones of a project and how they interact with each other.

DyeVC shows the information in different levels of detail, from a high-level topology-like visualization, where each node represents a repository clone, to a detailed level that presents every commit, despite the clone where it is located. The visualizations use transformations to present vertices and edges using different icons, colors, line types, and text labels, according to the characteristics that we want to highlight. This way, we established a framework for coupling different visualizations related to DVCS.

We have evaluated DyeVC on a real project, showing that it can be used to answer questions that arise when working with DVCSs. We have also performed an observation study that allowed subjects to use DyeVC to answer questions that are common when using DVCSs. Finally, we have also evaluated DyeVC's performance when used with repositories of different sizes, and we found out that the time and space complexity of the approach are directly related to the number of commits in the repository under analysis, especially in the view levels with finer granularity. The current version of DyeVC allows one to analyze repositories with different sizes with a limitation of 6,417 commits shown in the finest level of granularity (level 4, which shows the commit history), when using 2 GB of memory.

5.2 LIMITATIONS

DyeVC has a scalability limitation, regarding processing performance and memory usage, when dealing with level 4 information (commit history). We use the *Dijkstra's algorithm* (1959) provided by the JUNG graph library to minimize the number of crossing lines in the lower level visualization (that shows each commit in the topology). Although information from levels 1 to 3 is presented at reasonable times for repositories with tens of thousands of commits,

this procedure is not optimized to deal with graphs that contain more than 6,417 nodes in level 4. In order to draw these graphs, all vertex and edges must be loaded into memory to calculate vertices positions. Besides that, for repositories with more than 5,000 commits, more than a minute is spent to calculate positions and draw the graph. The usage of automatically collapsing could help in terms of time spent to draw the graph, as there would be less vertices and edges to be plot, but the memory issue would still be present, because collapsed nodes would be still loaded into memory, due to the limitation on how JUNG works. A possible way to solve that would be to filter commits before plotting them, for example showing only commits performed over the last month. A downside of this approach is that it could lead to a disconnected graph, for example, if work has been done over the last month on two separate branches whose common ancestor is a commit performed a long time ago, we would see two parallel sequence of commits, with no common ancestor.

Another limitation is related to the need of a central database to record information gathered from the several DyeVC instances. Although this central database is needed, we used a document-based database, and the information is read and written using semi-structured JSON documents, which are automatically mapped to/from the application class model. The connection with the central database is authenticated by using an application key. Although we do not store any sensitive information (we do not store contents of any files, just metadata), this might be a concern for some people. A different database and application key might be configured by editing the application configuration (more information regarding DyeVC usage and configuration is available in Appendix B).

The usage of a central database also brings a limitation related to the availability of the solution, because DyeVC visualizations need to access the central database to retrieve the information that will be shown. If the database goes down, DyeVC visualizations will not work properly, until it is available again. This limitation could be solved by replicating the central database locally, or by installing local databases together with each DyeVC instance and establishing a peer-to-peer synchronization among them. DyeVC also needs network connectivity in order to present visualizations. Having a local database would help to overcome this limitation as well, bringing the possibility of using DyeVC without a network (for example, while travelling on a plane).

An existing limitation in Level 2 visualization (which shows the topology) is regarding how the approach registers existing clones. In this visualization, once registered, clones will be presented forever. It might be the case that one had just registered a clone with DyeVC and

never worked on it again. After some time, this could lead to a polluted topology view, with lots of “garbage”, i.e., repositories that are not used or that might not even exist. The approach could check when was the last change in each clone, marking those clones that did not change for a period time, so that an administrator could remove it from the topology. Similarly, an administrator could manually include nodes in the topology, to represent clones located in places with no DyeVC instance running, in order to complete the topology not previously seen by the approach.

5.3 FUTURE WORK

The advent of DyeVC approach brings with it a number of possibilities for future researches. The following paragraphs describe possible improvements and researches that can be explored in the future.

The first improvement is related to the visualizations the approach already provides. Level 4 visualization, which shows every commit in the topology, could be enhanced with automatic collapsing of similar nodes. Currently, each vertex in level 4 visualization represents a single commit. Depending on the repository size, this leads to a graph that is very long horizontally, because we show each commit on a different X-coordinate, to give the idea of elapsed time. Even with the zooming feature, large repositories can be difficult to analyze. It happens that we normally want to analyze the very ending part of a repository, which comprises of the most recent commits in the topology, because the older ones probably were spread to the whole topology already. The current implementation has a feature for the user to select a group of commits and manually collapse them, creating a single node that represents the group of collapsed commits, which is placed at the midpoint between the first and the last collapsed nodes. However, on a repository with thousands of commits, this is not practical. Automatic collapsing could compact the visualization, by collapsing contiguous nodes that represent commits of the same type (pending to be pulled, pending to be pushed, synchronized, etc.) and leaving only branch heads expanded.

A number of research opportunities arise by increasing the amount of metadata that DyeVC already gathers,. For example, supposing that we are dealing with text artifacts, if DyeVC gathers the changes introduced by each commit at the line level (by storing each commit’s *diff*), one could create a visualization to show conflicts that will happen when merging two or more branches. This could be used to propose to the user an optimal sequence of merges, so that the number of conflicts is minimized, lowering the effort needed to perform the merges.

This additional data could also be used to mine information in the repositories, allowing uncovering usage patterns or presenting metrics. Mined information, together with the ability of creating new visualizations, could be helpful to answer a number of user questions, such as: Which repositories or people changed a specific artifact or group of artifacts? Which commits introduced the higher amount of changes in the code? Who were the top contributors in the project this week? Who were the top contributors in a file or in an application module? The answer to this last question could help, for example, in finding those developers who are experts on a module of the application.

BIBLIOGRAPHY

APPLETON, B.; BERZUK, S.; CABRERA, R.; ORENSTEIN, R. Streamed lines: Branching patterns for parallel software development. In: PATTERN LANGUAGES OF PROGRAMS CONFERENCE (PLOP 98), Aug. 1998, Monticello, Illinois, USA: ACM, Aug. 1998.

BATTIN, R. D.; CROCKER, R.; KREIDLER, J.; SUBRAMANIAN, K. Leveraging resources in global software development. *IEEE Software*, v. 18, n. 2, p. 70–77, Mar. 2001.

BIEHL, J. T.; CZERWINSKI, M.; SMITH, G.; ROBERTSON, G. G. FASTDash: A Visual Dashboard for Fostering Awareness in Software Teams. In: ACM CONFERENCE ON HUMAN FACTORS IN COMPUTING SYSTEMS (CHI '07), May 2007, New York, NY, USA: ACM, May 2007. p. 1313–1322.

BRUN, Y.; HOLMES, R.; ERNST, M. D.; NOTKIN, D. Proactive detection of collaboration conflicts. In: ACM SIGSOFT SYMPOSIUM AND EUROPEAN CONFERENCE ON FOUNDATIONS OF SOFTWARE ENGINEERING (ESEC/FSE'11), Sep. 2011, New York, NY, USA: ACM, Sep. 2011. p. 168–178.

CEDERQVIST, P. *Version Management with CVS*. [N.A.]: Free Software Foundation, 2005.

CESARIO, C. M.; MURTA, L. G. P. What is going on around my repository? In: BRAZILIAN WORKSHOP ON SOFTWARE VISUALIZATION, EVOLUTION AND MAINTENANCE (VEM'13), 29 Sep. 2013, Brasilia, Brazil: UNB, 29 Sep. 2013. p. 14–21.

CHACON, S. *Pro Git*. 1. ed. Berkeley, CA, USA: Apress, 2009.

CHODOROW, K. *MongoDB: The Definitive Guide*. 2. ed. Beijing: O'Reilly Media, 2013.

COLLBERG, C.; KOBOUROV, S.; NAGRA, J.; PITTS, J.; WAMPLER, K. A System for Graph-based Visualization of the Evolution of Software. In: ACM SYMPOSIUM ON SOFTWARE VISUALIZATION (SOFTVIS '03), Jun. 2003, New York, NY, USA: ACM, Jun. 2003. p. 77–ff.

COLLINS-SUSSMAN, B.; FITZPATRICK, B. W.; PILATO, C. M. *Version Control with Subversion*. Stanford, CA, USA: Compiled from r4849, 2011.

DA SILVA, I. A.; CHEN, P. H.; VAN DER WESTHUIZEN, C.; RIPLEY, R. M.; VAN DER HOEK, A. Lighthouse: coordination through emerging design. In: WORKSHOP ON

ECLIPSE TECHNOLOGY EXCHANGE (OOPSLA '06), Oct. 2006, New York, NY, USA: ACM, Oct. 2006. p. 11–15.

DEWAN, P.; HEGDE, R. Semi-synchronous conflict detection and resolution in asynchronous software development. In: EUROPEAN CONFERENCE ON COMPUTER-SUPPORTED COOPERATIVE WORK (ECSCW '07), Sep. 2007, Limerick, Ireland: Springer London, Sep. 2007. p. 159–178.

DIEHL, S. *Software Visualization: Visualizing the Structure, Behaviour, and Evolution of Software*. Berlin; New York: Springer, 2007.

DIJKSTRA, E. W. A note on two problems in connexion with graphs. *Numerische Mathematik*, v. 1, n. 1, p. 269–271, 1 Dec. 1959.

DONG, J.; GANG, X. A Topology Discovery Algorithm Based on the IP-Network. In: 2012 INTERNATIONAL CONFERENCE ON CONTROL ENGINEERING AND COMMUNICATION TECHNOLOGY (ICCECT'12), Dec. 2012, Shenyang, Liaoning, China: IEEE, Dec. 2012. p. 665–668.

DOURISH, P.; BELLOTTI, V. Awareness and Coordination in Shared Workspaces. In: ACM CONFERENCE ON COMPUTER-SUPPORTED COOPERATIVE WORK (CSCW '92), Nov. 1992, New York, NY, USA: ACM, Nov. 1992. p. 107–114.

ECLIPSE FOUNDATION. *The Open Source Developer Report - 2013 Eclipse Community Survey*. Survey. San Francisco, CA, USA: Eclipse Foundation, Jun. 2013.

ELLIOTT, J.; ECKSTEIN, R.; LOY, M.; COLE, B. *Java Swing, Second Edition*. 2. ed. Sebastopol, CA: O'Reilly Media, 2002.

ELSEN, S. VisGi: Visualizing Git branches. In: IEEE WORKING CONFERENCE ON SOFTWARE VISUALIZATION (VISSOFT'13), Sep. 2013, Eindhoven, Netherlands: IEEE, Sep. 2013. p. 1–4.

ESTUBLIER, J. Software configuration management: a roadmap. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING (ICSE '00), May 2000, New York, NY, USA: ACM, May 2000. p. 279–289.

FIELDING, R. T. *Architectural Styles and the Design of Network-based Software Architectures*. 2000. Thesis – University of California, Irvine, CA, USA, 2000.

FITZPATRICK, G.; MARSHALL, P.; PHILLIPS, A. CVS Integration with Notification and Chat: Lightweight Software Team Collaboration. In: ACM CONFERENCE ON COMPUTER-SUPPORTED COOPERATIVE WORK (CSCW '06), Nov. 2006, New York, NY, USA: ACM, Nov. 2006. p. 49–58.

FUKS, H.; RAPOSO, A.; GEROSA, M. A.; PIMENTEL, M.; LUCENA, C. J. The 3c collaboration model. In: KOCK, N. (Org.). . *The Encyclopedia of E-Collaboration*. New York, NY, USA: Information Science Reference, 2007. p. 637–644.

GILBERT, E.; KARAHALIOS, K. LifeSource: Two CVS Visualizations. In: ACM CONFERENCE ON HUMAN FACTORS IN COMPUTING SYSTEMS (CHI '06), Apr. 2006, New York, NY, USA: ACM, Apr. 2006. p. 791–796.

GUIMARÃES, M. L.; SILVA, A. R. Improving early detection of software merge conflicts. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING (ICSE '12), Jun. 2012, Piscataway, NJ, USA: IEEE Press, Jun. 2012. p. 342–352.

GUMM, D.-C. Distribution Dimensions in Software Development Projects: A Taxonomy. *IEEE Software*, v. 23, n. 5, p. 45–51, Sep. 2006.

GUTWIN, C.; GREENBERG, S.; ROSEMAN, M. Workspace Awareness in Real-Time Distributed Groupware: Framework, Widgets, and Evaluation - Springer. In: SASSE, M. A.; CUNNINGHAM, R. J.; WINDER, R. L. (Org.). . *People and Computers XI*. London: Springer London, 1996. p. 281–298.

HOZUMI, T. *Visugit*. Available at: <<https://github.com/hozumi/visugit>>. Accessed: 2 jan. 2015.

LANZA, M. The Evolution Matrix: Recovering Software Evolution Using Software Visualization Techniques. In: INTERNATIONAL WORKSHOP ON PRINCIPLES OF SOFTWARE EVOLUTION (IWPSE '01), Sep. 2001, New York, NY, USA: ACM, Sep. 2001. p. 37–42.

LEON, A. *Software Configuration Management Handbook, Second Edition*. 2. ed. Norwood, MA, USA: Artech House, 2004.

LI, H.; DAN, C.; HUAIXIANG, B.; SHURONG, L. Topology Discovery Algorithm Based on Ant Colony Algorithm of Power Line Carrier Sensor Network. In: INTERNATIONAL CONFERENCE ON COMMUNICATION SOFTWARE AND NETWORKS (ICCSN '09), Feb. 2009, Macau, China: IEEE, Feb. 2009. p. 102–105.

LI, M.; YANG, J.; AN, C.; LI, C.; LI, F. IPv6 network topology discovery method based on novel graph mapping algorithms. In: IEEE SYMPOSIUM ON COMPUTERS AND COMMUNICATIONS (ISCC '13), Jul. 2013, Split, Croatia: IEEE, Jul. 2013. p. 554–560.

MARINILLI, M. *Java Deployment with JNLP and WebStart*. 1. ed. Indianapolis, Ind: Sams Publishing, 2001.

MURTA, L. G. P. *Gerência de Configuração no Desenvolvimento Baseado em Componentes*. 2006. Thesis – UFRJ, COPPE, Rio de Janeiro, Brasil, 2006.

MURTA, L. G. P. *Version Control - an Introduction, lecture notes distributed in Software Configuration Management Laboratory at Universidade Federal Fluminense*. Niteroi, RJ, Brazil, 17 Aug. 2012.

O’SULLIVAN, B. Making sense of revision-control systems. *Communications of the ACM*, v. 52, n. 9, p. 56–62, Sep. 2009a.

O’SULLIVAN, B. *Mercurial: The Definitive Guide*. 1. ed. Sebastopol, CA, USA: O’Reilly Media, 2009b.

PEARSON, K. Note on Regression and Inheritance in the Case of Two Parents. *Proceedings of the Royal Society of London*, v. 58, n. 347-352, p. 240–242, 1 Jan. 1895.

PERRY, D. E.; SIY, H. P.; VOTTA, L. G. Parallel changes in large scale software development: an observational case study. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING (ICSE 98’), Apr. 1998, Washington, DC, USA: IEEE Computer Society, Apr. 1998. p. 251–260.

PRESTON-WERNER, T. *GitHub’s Network Graph*. Available at: <<https://github.com/blog/39-say-hello-to-the-network-graph-visualizer>>. Accessed: 2 jan. 2015.

ROCHKIND, M. J. The source code control system. *IEEE Transactions on Software Engineering. (TSE)*, v. 1, n. 4, p. 364–470, Dec. 1975.

SANTOS, R.; MURTA, L. G. P. Evaluating the Branch Merging Effort in Version Control Systems. In: BRAZILIAN SYMPOSIUM ON SOFTWARE ENGINEERING (SBES ’12), Sep. 2012, Natal, RN - Brazil: IEEE Computer Society, Sep. 2012. p. 151–160.

SARMA, A.; REDMILES, D. F.; VAN DER HOEK, A. Palantir: Early Detection of Development Conflicts Arising from Parallel Code Changes. *IEEE Transactions on Software Engineering*, v. 38, n. 4, p. 889–908, Aug. 2012.

SCHOLLMEIER, R. A definition of peer-to-peer networking for the classification of peer-to-peer architectures and applications. In: INTERNATIONAL CONFERENCE ON PEER-TO-PEER COMPUTING (P2P’01), Aug. 2001, Linkoping, Sweden: IEEE, Aug. 2001. p. 101–102.

SHULL, F.; CARVER, J.; TRAVASSOS, G. H. An Empirical Methodology for Introducing Software Processes. In: EUROPEAN SOFTWARE ENGINEERING CONFERENCE HELD JOINTLY WITH ACM SIGSOFT INTERNATIONAL SYMPOSIUM

ON FOUNDATIONS OF SOFTWARE ENGINEERING (ESEC/FSE-9), 2001, New York, NY, USA: ACM, 2001. p. 288–296.

STEINMACHER, I.; CHAVES, A.; GEROSA, M. Awareness Support in Distributed Software Development: A Systematic Review and Mapping of the Literature. *15th ACM Conference on Computer-supported Cooperative Work (CSCW '12)*, p. 1–46, May 2012.

SVAHNBERG, M.; AURUM, A.; WOHLIN, C. Using Students As Subjects - an Empirical Evaluation. In: INTERNATIONAL SYMPOSIUM ON EMPIRICAL SOFTWARE ENGINEERING AND MEASUREMENT (ESEM '08), Oct. 2008, New York, NY, USA: ACM, Oct. 2008. p. 288–290.

TICHY, W. RCS: A system for version control. *Software - Practice and Experience*, v. 15, n. 7, p. 637–654, 1985.

UZAIR, U.; AHMAD, H. F.; ALI, A.; SUGURI, H. An Efficient Algorithm for Ethernet Topology Discovery in Large Multi-subnet Networks. In: IEEE INTERNATIONAL CONFERENCE ON SYSTEM OF SYSTEMS ENGINEERING (SOSE '07), Apr. 2007, San Antonio, TX, USA: IEEE, Apr. 2007. p. 1–7.

VOINEA, L.; TELEA, A.; VAN WIJK, J. J. CVSscan: Visualization of Code Evolution. In: ACM SYMPOSIUM ON SOFTWARE VISUALIZATION (SOFTVIS '05), May 2005, New York, NY, USA: ACM, May 2005. p. 47–56.

WALRAD, C.; STROM, D. The importance of branching models in SCM. *IEEE Computer*, v. 35, n. 9, p. 31 – 38, Sep. 2002.

WEBSTER, J.; WATSON, R. T. Analyzing the past to prepare for the future: Writing a literature review. *Management Information Systems Quarterly*, v. 26, n. 2, p. 3, 2002.

YAN, H. The study on network topology discovery algorithm based on SNMP protocol and ICMP protocol. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING AND SERVICE SCIENCE (ICSESS '12), Jun. 2012, Beijing, China: IEEE, Jun. 2012. p. 665–668.

YONG, W.; NAN, P.; XIAOLING, T. Network topology discovery algorithm based on OSPF. In: INTERNATIONAL CONFERENCE ON INTELLIGENT COMPUTING AND INTEGRATED SYSTEMS (ICISS '10), Oct. 2010, Guilin, China: IEEE, Oct. 2010. p. 136–139.

APPENDIX A – COMMIT HISTORY VISUALIZATION

As discussed in Section 3.3.4, Level 4 information consists in a visualization that shows all commits pertaining to a project as a DAG, where each vertex represents a commit. This appendix presents more information regarding how this visualization is built.

We created a layout to plot the graph, named *RepositoryHistoryLayout*, which takes a JUNG graph as input. A JUNG graph is implemented as a double linked list l , where each vertex v is an element of l . Thus, for a commit represented by v , it is possible to get both the predecessors (parents) and successors (children) of v . For example, Figure 47 shows that commit 3 has one predecessor (2) and two successors (4 and 5).

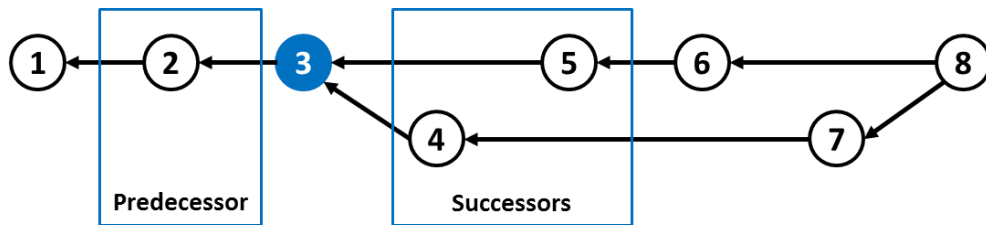


Figure 47 – Predecessors and successors for a commit

In order to plot a graph, we have to calculate the x and y positions for every vertex v in the graph. To calculate the x position, we start from the first commit in the repository. This is not necessarily the commit with the earliest commit date because, as we discussed in Section 3.3.4, DVCSs do not have the concept of a central clock. To find out the first commit, we look for the commit that has no predecessors and assign 0 (zero) to its x position (This would be commit 1 in Figure 47). After that, we take all its successors (children commits) and, for each one of them, we look at the commit date to find out which one is the earliest, assigning greater values for x as the algorithm proceeds, until all commits have been assigned a value for x . If we find that a commit has a child with lower value for x value then itself, this is due to a clock problem, and we correct it by changing both commits x values. This is the case shown in Figure 48, where commit 8 should be to the right of commit 7. Notice that the edge that connects commit 8 to commit 7 is in the wrong direction.

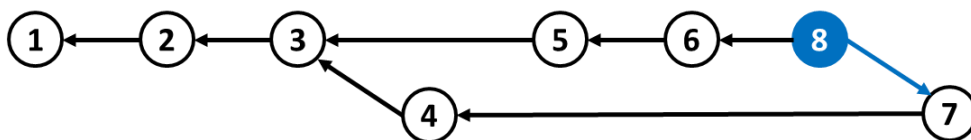


Figure 48 – Commit with wrong value for x

In order to calculate the y position of each node, processing starts from the farthest commit to the right, for which a value of 0 is assigned to its y position. This is to guarantee that the branch with the most recent commit in the history always appear at the top. Figure 49 presents an example of a small commit history showing the y positions, beginning in 0, down to -100. It is possible to notice that most recent commit is A, and that its y position is 0. Other branches are assigned values below 0 for y position (branches headed by commits B and C were assigned y positions -60 and -100, respectively), which makes the graph grow to the bottom.

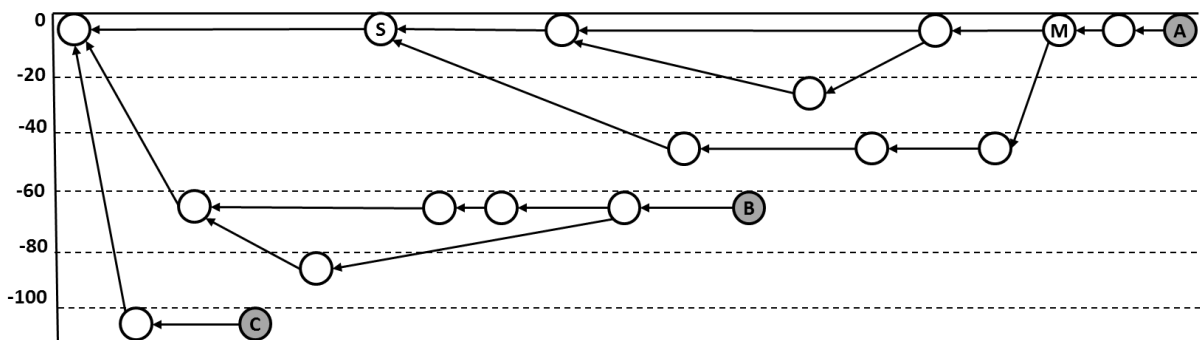


Figure 49 – Calculating y position

The calculation occurs from right to left, looking at the predecessors of each commit. When a commit has more than one predecessor (like commit *M* in Figure 49, this because it is a merge. To find out the y position for each of *M*'s predecessors, it is necessary to find their common ancestor (which is *S* in our example) and calculate the height of the subtree between these two commits (*S* and *M*). In this example, the height of the subtree between *S* and *M* is two. Therefore, one of *M*'s predecessors is assigned 0 for y position (same position as *M*), while the other is assigned -40 (which is two steps down). There could be more merges between *S* and *M* and the process continues until all nodes are visited and their y position is properly set.

APPENDIX B – DYEVC USAGE

B.1 INTRODUCTION

This Appendix show basic usage information regarding DyeVC, together with some tips on its configuration. This information is also available online on the DyeVC User Manual³².

B.2 RUNNING DYEVC

As we discussed in Section 3.5, DyeVC uses Java Web Start technology and thus does not need to be formally installed. After launching the application for the first time, it creates a shortcut in the Desktop (Figure 50.a), which can be used to execute the application later on. After running the application, it lies on the system tray bar (Figure 50.b). A single click on the icon will show the application window and minimizing it will take it back to the tray bar.

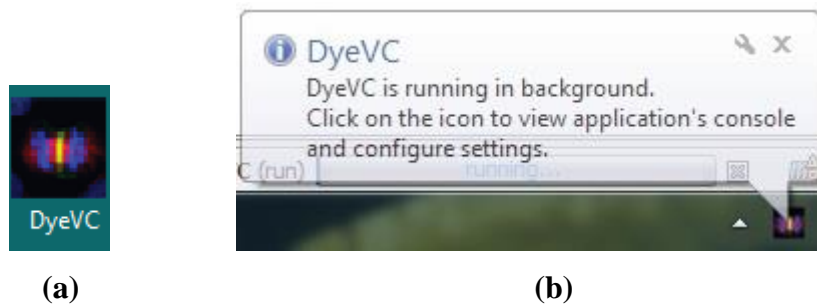


Figure 50 – DyeVC icon on the desktop and on the tray bar

B.2.1 MAIN WINDOW

After maximizing the application, the main window is shown (Figure 51). The main window presents all monitored repositories, along with the following information in the Monitored Repositories panel:

- **Status:** An icon representing the clone status related to its known partners (as discussed in Section 3.3);
- **System Name:** The system or project name that the clone belongs. Clones that belong to the same project are shown together in the topology view;
- **Clone Name:** The name that the user gave to this particular clone. It must be unique on each single machine for a particular system name;
- **Id:** An internal unique id generated by DyeVC;

³² <https://github.com/gems-uff/dyevc/wiki/User-Manual>

- **Clone Path:** The path in the local machine where this clone is found.

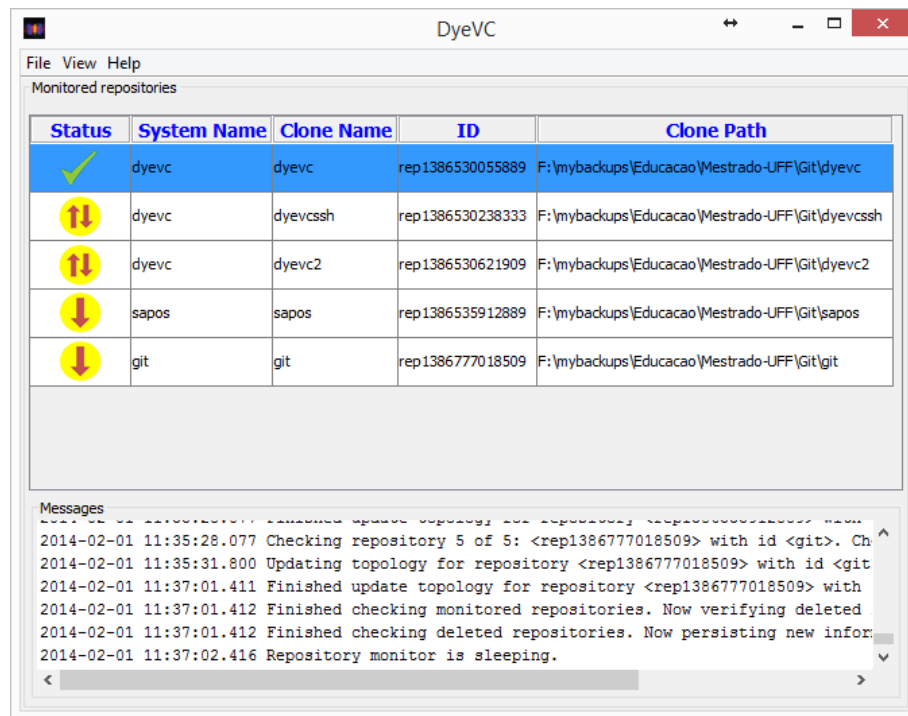


Figure 51 – DyeVC main window

The application also shows relevant information regarding the monitoring status in the Messages panel. There is the possibility to see more detailed log messages by clicking on **View → Console Window**. A new window will be opened, where messages are displayed according to their levels of criticality. The default behavior is to show INFO, WARN, and ERROR messages, but right clicking on the panel allows the user to change this behavior, allowing to also log TRACE and DEBUG messages.

B.2.2 MONITORED REPOSITORIES

The main screen will be initially empty as there is no repository being monitored. Clicking on **File → Add Project** allows creating a new monitoring configuration (Figure 52). The user can choose a system name from the ones provided on the drop-down list, or type a new one, and click on the Explore button to choose the path where the clone is located. The Clone Name will be automatically chosen by the application, based on the folder name where the clone is located. For instance, if the user points the Clone Address to */home/users/username/myprojects/xyz*, the Clone Name of this configuration will be *xyz*.

Creating a new monitoring configuration

System Name:

Clone Name:

Clone Address:

Figure 52 – Creating a new monitoring configuration

B.2.3 VISUALIZATIONS

Once repositories are being monitored, the user is able to navigate through all the visualization levels discussed in Section 3.3, where each one of them is described. Level 1 (Notifications) will be shown as notifications in tray bar (Figure 50.b); Level 2 (Topology) will be presented by right clicking on a repository and choosing Show Topology (Figure 53); Level 3 (Tracked Branches) will be shown by hovering the mouse over any monitored repository (Figure 51); and Level 4 (Commits) will be accessible by right clicking on a repository and choosing Show Commit History (Figure 54).

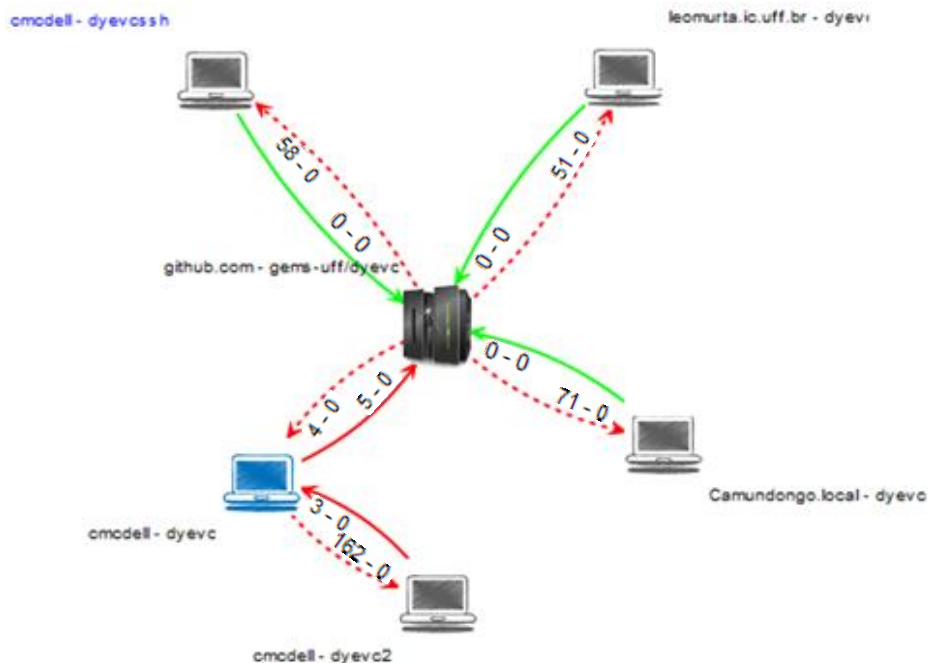


Figure 53 – Topology view for a given project

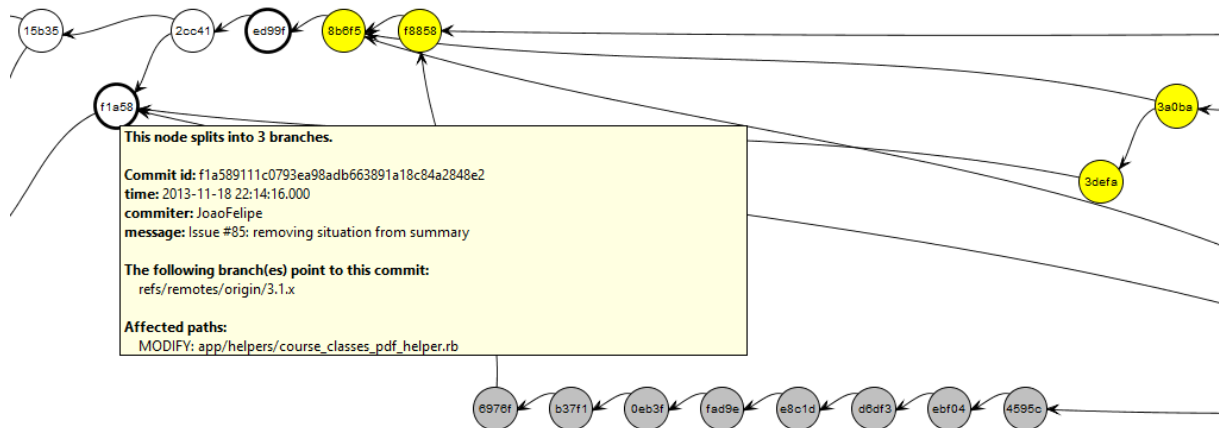


Figure 54 – Commit history for a given project

B.3 TYPICAL USAGES

DyeVC visualizations may help both administrators and developers that work with DVCS. However, each profile may use DyeVC differently. Figure 55 represents the typical DyeVC usage by administrators. After starting DyeVC, an administrator will typically maximize it and invoke any of its visualizations. After working with the desired visualizations, the administrator will then stop DyeVC.

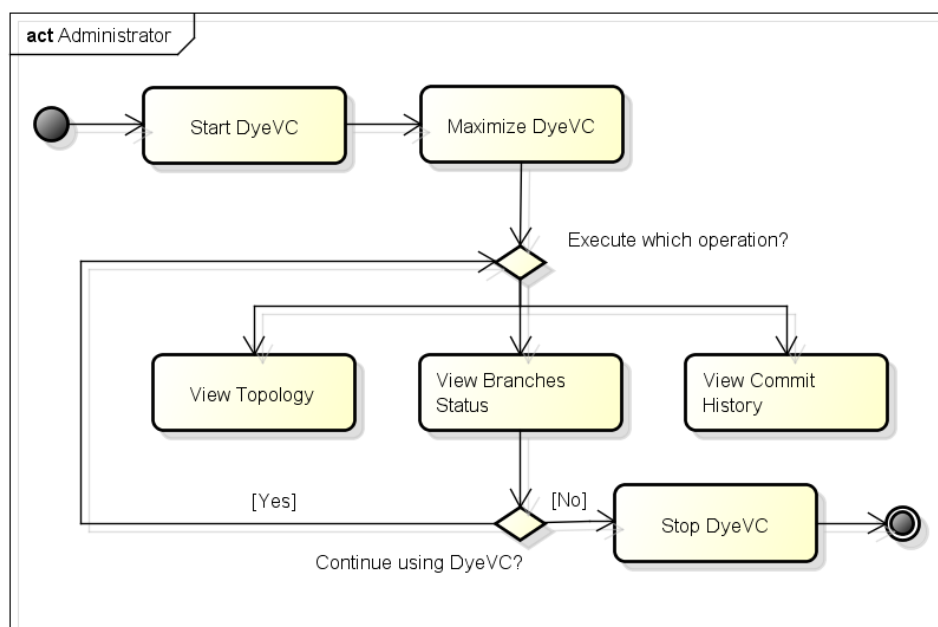


Figure 55 – Typical DyeVC usage by administrators

Figure 56 represents the typical DyeVC usage by developers. After starting DyeVC, a developer will generally leave it minimized. After DyeVC sends a notification, the developer may decide to check more details of the notification, in which case they would maximize it and

invoke any of its visualizations. After working with the desired visualizations, the developer would minimize it again, until another notification is received and they decide to check it. The developer could also stop DyeVC, but this would generally happen only at the end of a workday.

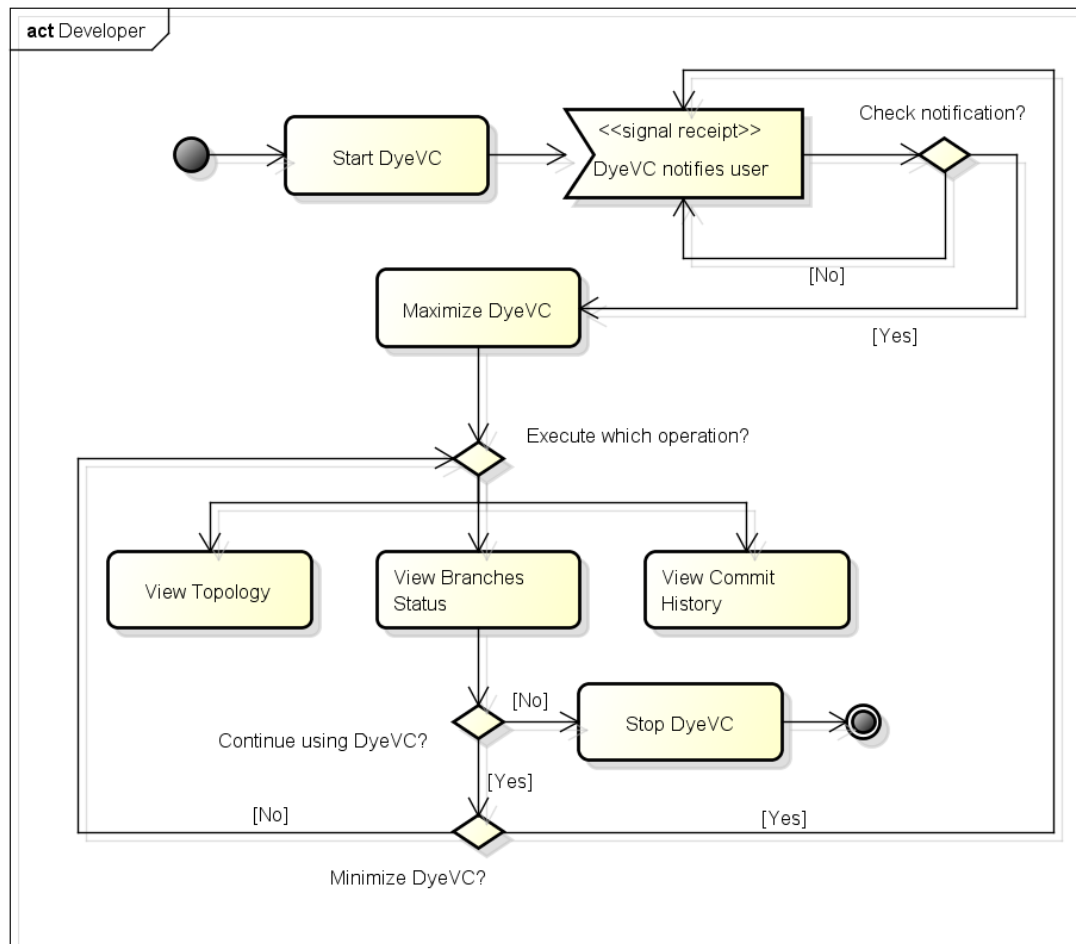


Figure 56 – Typical DyeVC usage by developers

B.4 FURTHER CONFIGURATIONS

This Section discusses further configurations that may be needed to adjust DyeVC to the user's need.

B.4.1 REFRESH RATE

DyeVC periodically updates the state of all monitored repositories. The time elapsed between subsequent updates is configured through the **DyeVC Settings** window (Figure 57). This setting is in seconds and its default value is 300, meaning that the application will check the repositories at each 5 minutes.

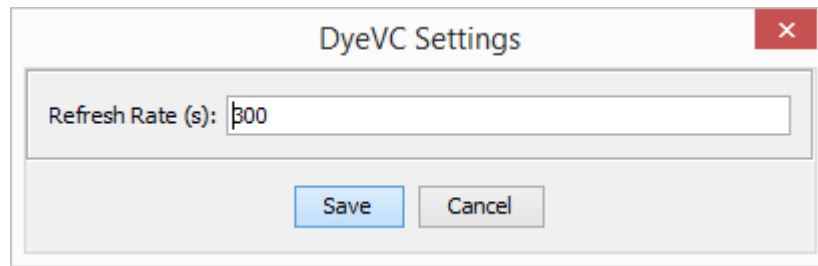


Figure 57 – DyeVC settings window

B.4.2 CONNECTING TO A DIFFERENT DATABASE

If user wants to use a different database to store topology data, it is necessary to manually create or edit a preferences key, as described below, according to the user's operating system:

- On Windows systems, use the regedit tool and create / edit the following keys under HKEY_CURRENT_USER\Software\JavaSoft\Prefs\br\uff\ic\dyevc\application\generalsettings (both keys must be created as String values):
 - **databasePath**: this key stores the URL to the MongoDB instance;
 - **appKey**: this key stores the application key used to access MongoDB REST services.
- On Linux or Mac systems, create the following text files under ~/Library/Preferences/br.uff.ic.dyevc.application/generalsettings:
 - **databasePath**: this file must contain the URL to the MongoDB instance;
 - **appKey**: this file must contain the application key used to access MongoDB REST services.

B.4.3 CLEARING THE CACHE

If any visualization does not match the user's environment, this can possibly due to a bug in the application. Although this bug may have already been fixed, DyeVC stores some cache information to speed up the analysis, and this cache must be cleared in order to fix the visualization.

To clear the cache, the user must right click on the desired monitored repository and then on **Clear Cache and Check Project**. This will clear the cache for that project and force DyeVC to check it again as if it was just added to the list of monitored repositories.

APPENDIX C – INFORMED CONSENT FORM

TERMO DE CONSENTIMENTO LIVRE E ESCLARECIDO (TCLE)

Condutor do Estudo: Cristiano Machado Cesário (aluno de mestrado)

Pesquisador Responsável: Prof. Leonardo Murta – {leomurta}@ic.uff.br

Instituição: Universidade Federal Fluminense (UFF)

Eventualmente realizamos estudos experimentais para caracterizar/avaliar uma determinada tecnologia de software. Estes estudos são conduzidos por alunos de Pós-graduação em Computação da Universidade Federal Fluminense (UFF). Você foi previamente selecionado pelo seu perfil/conhecimento/experiência e está sendo convidado a participar desta pesquisa. Essa pesquisa consiste em avaliar o apoio fornecido por ferramentas de software (DyeVC e/ou Git) na percepção de atualizações em projetos de software.

1) Procedimentos

O estudo está sendo realizado com data e hora marcada com os participantes pré-selecionados. O estudo será executado de forma individual. O estudo consiste no uso da ferramenta DyeVC e/ou Git para responder perguntas relacionadas ao andamento do projeto JQuery. Durante o estudo é esperado que você expresse em voz alta o que está pensando/fazendo. O áudio será gravado e transcrito para viabilizar a análise posterior da sessão. Caso seja necessário, ao final do estudo será solicitado que você responda um questionário de avaliação sobre a tecnologia de software que está sendo caracterizada/avaliada.

2) Tratamento de possíveis riscos e desconfortos

Serão tomadas todas as providências durante a coleta de dados de forma a garantir a sua privacidade e seu anonimato.

3) Benefícios e Custos

Espera-se que, como resultado deste estudo, você possa aumentar seus conhecimentos, de maneira a contribuir para o aumento da qualidade das atividades com as quais você trabalhe ou possa vir a trabalhar. Este estudo também contribuirá com resultados importantes para a pesquisa de um modo geral. Você não terá nenhum gasto ou ônus com a sua participação no estudo e também não receberá qualquer espécie de reembolso ou gratificação devido à autorização do uso dos dados coletados nesse estudo.

4) Confidencialidade da Pesquisa

Toda informação coletada neste estudo é confidencial e seu nome não será identificado de modo algum. Quando os dados forem coletados, seu nome será removido dos mesmos e não será utilizado em nenhum momento durante a análise ou apresentação dos resultados.

5) Participação

Sua participação neste estudo é muito importante e voluntária, pois requer a sua aprovação para utilização dos dados coletados. Você tem o direito de não querer participar ou de sair deste estudo a qualquer momento, sem penalidades. Em caso de você decidir se retirar do estudo, favor notificar o pesquisador responsável. Você pode solicitar e esclarecimento sobre o estudo a qualquer momento.

6) Declaração de Consentimento

Declaro que li e estou de acordo com as informações contidas neste documento e que toda linguagem técnica utilizada na descrição deste estudo de pesquisa foi explicada satisfatoriamente, recebendo respostas para todas as minhas dúvidas. Confirmando também que recebi uma cópia deste Termo (TCLE), compreendo que sou livre para não autorizar a utilização dos meus dados neste estudo em qualquer momento, sem qualquer penalidade. Declaro ter mais de 18 anos e concordo de espontânea vontade em participar deste estudo.

Data:

Nome do Participante (letra de forma):

RG do Participante:

Assinatura:

APPENDIX D – CHARACTERIZATION FORM

Questionário de Caracterização

Este formulário contém algumas perguntas sobre sua experiência acadêmica e profissional.

1. Formação acadêmica

- ☐ Doutorado
- ☐ Doutorando
- ☐ Mestrado
- ☐ Mestrando
- ☐ Graduação

Ano de ingresso: _____ Ano de conclusão (ou previsão de conclusão): _____

2. Formação Geral

2.1. Em que tipo de projetos você ocupa a maior parte do tempo?

- a) Acadêmicos
- b) Pessoais
- c) Open-Source
- d) Indústria

2.2. Quantos anos de experiência em programação você possui?

- e) 0-2 anos
- f) 3-5 anos
- g) 6-10 anos
- h) Mais de 10 anos

2.3. Qual SCV você utiliza com mais frequência?

- i) Git
- j) Subversion
- k) CVS
- l) Mercurial
- m) Outro. Especificar: _____

2.4. Com quantas pessoas, em média, você costuma trabalhar em equipes de desenvolvimento?

- n) Apenas eu
- o) 2-5 pessoas
- p) 6-10 pessoas
- q) Mais de 10 pessoas

3. Utilização de Sistemas de Controle de Versão (SCV)

3.1. Com que frequência você efetua commit de suas mudanças?

- a) Depende
- b) A cada grupo de linhas editadas
- c) Ao terminar as alterações em um método de uma classe
- d) Ao terminar a *feature* que estou implementando ou o *bug* que estou corrigindo
- e) Uma vez por dia, antes de terminar o trabalho
- f) Quando lembro, ou quando alguém solicita que disponibilize as alterações que fiz

3.2. Se você escolheu “Depende”, sua decisão depende de que fatores?

3.3. Ao efetuar um commit, como você agrupa (ou quebra) suas alterações?

APPENDIX E – ACTIVITIES – PHASE 1

Estudo Observacional – Etapa 1

Instruções:

Este é um estudo de observação, por isso, sempre que possível, verbalize seus pensamentos, para que o experimentador possa melhor avaliar os resultados obtidos. Pergunte e comente tudo que achar necessário. Você terá 15 minutos para atuar em cada um dos cenários. Caso não consiga responder a algumas das perguntas, registre o fato e passe à pergunta seguinte.

Contextualização

Você trabalha em um projeto *open source* (jQuery) que recebe contribuições de desenvolvedores em todo o mundo. Por esse motivo, você não tem contato direto com as demais pessoas que contribuem para esse projeto.

Cenário 1:

Você é um desenvolvedor que está na trabalhando no projeto JQuery, em um clone chamado *aakoch*. Responda às seguintes questões:

1.1 Qual a situação de seu clone em relação ao repositório central?

- ☐ Sincronizado
- ☐ Adiantado em ____ commits
- ☐ Atrasado em ____ commits

1.2 Quem mais está trabalhando no projeto JQuery, além de você? (outros clones)

1.3 Quais foram os arquivos modificados no commit com hash iniciado em 5d454?

Cenário 2:

Após alguns meses, você passou a gerenciar o projeto JQuery. Responda às seguintes questões:

2.1 Quais são os clones existentes do JQuery?

2.2 Qual(is) clone(s) está(ão) sincronizado(s) com o repositório central?

2.3 Quantos commits em ramos rastreados estão pendentes de envio ao repositório central?

2.4 Existe algum commit realizado em ramo não rastreado? Onde?

APPENDIX F – ACTIVITIES – PHASE 2

Estudo Observacional – Etapa 2

Instruções:

Este é um estudo de observação, por isso, sempre que possível, verbalize seus pensamentos, para que o experimentador possa melhor avaliar os resultados obtidos. Pergunte e comente tudo que achar necessário. Você terá 15 minutos para atuar em cada um dos cenários. Caso não consiga responder a algumas das perguntas, registre o fato e passe à pergunta seguinte

Abordagem DyeVC

DyeVC é uma abordagem cujo objetivo é proporcionar a percepção de alterações (*awareness*) realizadas em repositórios de controle de versão distribuídos (DVCS – *Distributed Version Control Systems*). A abordagem consiste em um conjunto de visualizações, em diferentes níveis de detalhe, que proporcionam que os envolvidos em projetos que utilizam DVCS possam:

- Receber notificações na barra de tarefas, sempre que uma alteração ocorrer nos repositórios associados aos clones em que está trabalhando (i.e. repositórios para os quais faz push e dos quais faz pull);
- Visualizar os clones conhecidos de um projeto, e as dependências entre eles (i.e. quem se comunica com quem);
- Visualizar informações sobre os ramos rastreados, bem como sua situação em relação aos ramos correspondentes no(s) repositório(s) de origem;
- Visualizar o grafo de commits de toda a topologia, apresentando não só os commits existentes localmente, mas também aqueles commits que existam em outros repositórios (mesmo que não haja uma comunicação entre eles. Commits existentes em ramos não rastreados também são apresentados nessa visualização.

A Fig. 1 apresenta a tela principal do DyeVC e a Tabela 1 apresenta as possíveis situações de um repositório que são apresentadas nessa tela.

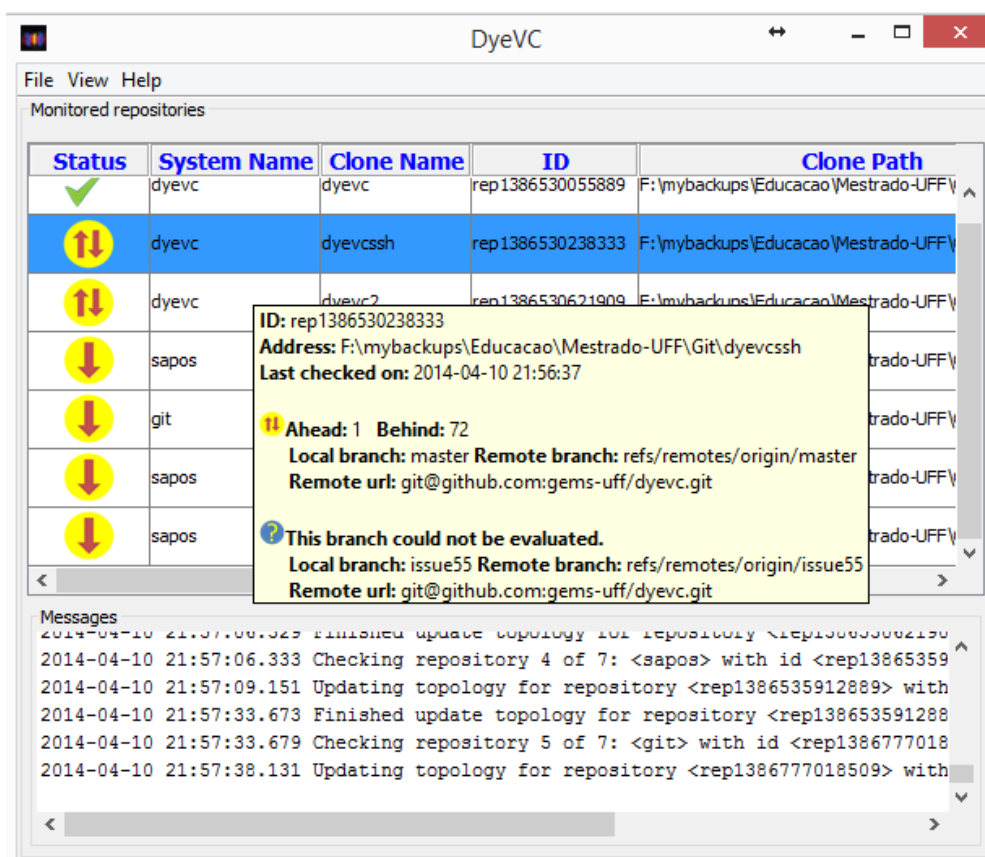


Fig. 1 – Tela principal do DyeVC

Tabela 1 – Possíveis status de um repositório

Status	Description
	DyeVC has not analyzed the repository yet.
	Repository is synchronized with all peers.
	Repository has changes that were not sent yet to its peers (it is ahead its peers).
	Peers have changes that were not sent yet to the repository (it is behind its peers).
	Repository is both ahead and behind its peers.
	Invalid repository. This happens when DyeVC cannot access the repository. The reason is presented to the user.

A Fig. 2 e a Fig. 3 apresentam os principais elementos da visualização de topologia do DyeVC.

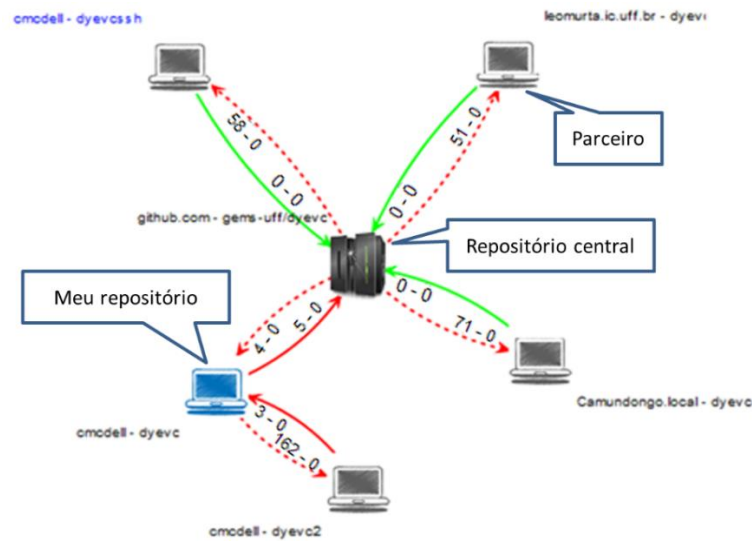


Fig. 2 – Elementos da visualização de topologia do DyeVC

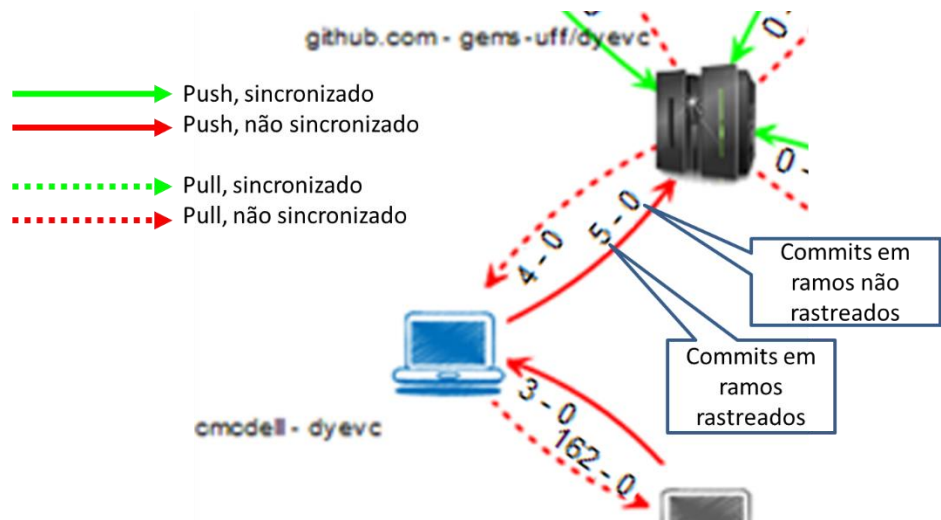


Fig. 3 – Elementos da visualização de topologia do DyeVC

A Fig. 4 apresenta a visão de commits da abordagem. O código de cores utilizado na representação de commits é apresentado na Fig. 5.

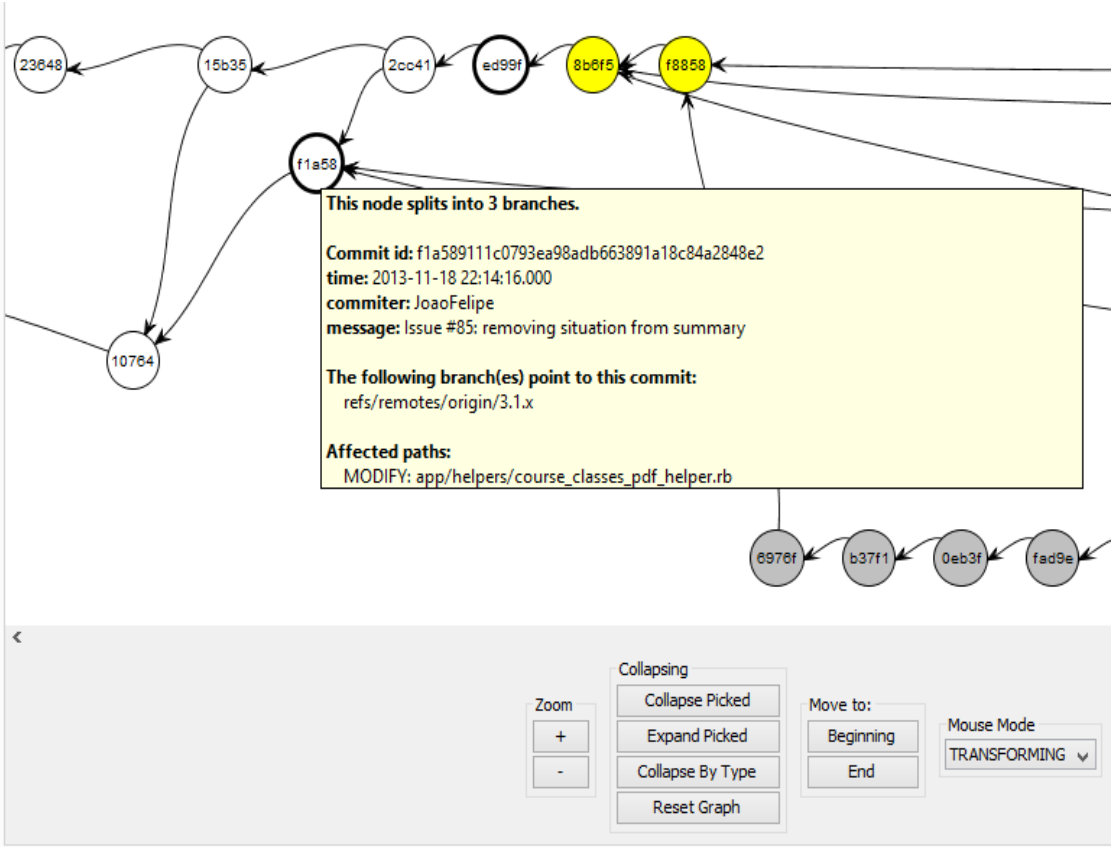


Fig. 4 – Visão de commits (história do repositório)

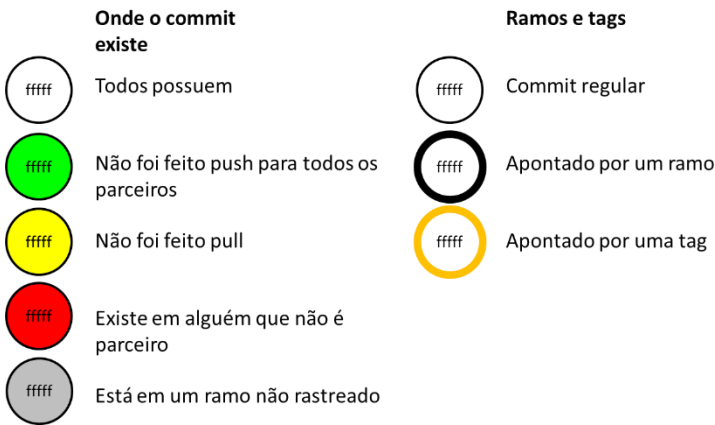


Fig. 5 – Elementos da visualização de commits do DyeVC

Contextualização

Após ter contato com a abordagem DyeVC, refaça os cenários 1 e 2 da etapa 1, verificando se sua resposta se mantém. Caso a resposta agora seja outra, ou caso não tenha conseguido responder à questão na primeira etapa e consiga agora, informe a nova resposta no formulário.

Cenário 1:

Você está trabalhando no projeto JQuery, em um clone chamado *aakoch*. Responda às seguintes questões:

1.1 Qual a situação de seu clone em relação ao repositório central?

<input type="checkbox"/> Mesma resposta da etapa 1	Resposta diferente: <input type="checkbox"/> Sincronizado <input type="checkbox"/> Adiantado em ____ commits <input type="checkbox"/> Atrasado em ____ commits
--	---

1.2 Quem mais está trabalhando no projeto JQuery, além de você? (outros clones)

<input type="checkbox"/> Mesma resposta da etapa 1	Resposta diferente:
--	---------------------

1.3 Quais foram os arquivos modificados no commit com hash iniciado em 5d454?

<input type="checkbox"/> Mesma resposta da etapa 1	Resposta diferente:
--	---------------------

Cenário 2:

Após alguns meses, você passou a gerenciar o projeto JQuery. Responda às seguintes questões:

2.1 Quais são os clones existentes do JQuery?

<input type="checkbox"/> Mesma resposta da etapa 1	Resposta diferente:
--	---------------------

2.2 Qual(is) clone(s) está(ão) sincronizado(s) com o repositório central?

<input type="checkbox"/> Mesma resposta da etapa 1	Resposta diferente:
--	---------------------

2.3 Quantos commits em ramos rastreados estão pendentes de envio ao repositório central?

<input type="checkbox"/> Mesma resposta da etapa 1	Resposta diferente:
--	---------------------

2.4 Existe algum commit realizado em ramo não rastreado? Onde?

<input type="checkbox"/> Mesma resposta da etapa 1	Resposta diferente:
--	---------------------

APPENDIX G – EXIT SURVEY

Nome:

Por favor, preencha a seguinte pesquisa sobre este experimento utilizando uma escala de 1 até 5 (onde 1 discorda plenamente, 2 discorda, 3 neutro, 4 concorda, 5 concorda plenamente ou N/A não se aplica)

Você achou fácil a interação com o DyeVC?

1 2 3 4 5 N/A

Você achou fácil identificar os repositórios relacionados no DyeVC?

1 2 3 4 5 N/A

Você achou fácil a utilização das operações disponibilizadas no DyeVC?

1 2 3 4 5 N/A

Qual operação disponibilizada pelo DyeVC você achou mais útil?

Você achou que as visualizações disponibilizadas pelo DyeVC foram úteis para responder as questões?

1 2 3 4 5 N/A

Qual visualização disponibilizada pelo DyeVC você achou mais útil?

Você acha que o uso do DyeVC lhe ajudou durante a investigação do projeto JQuery?

1 2 3 4 5 N/A

Liste os aspectos positivos da abordagem

Liste os aspectos negativos da abordagem

Você tem algum outro comentário sobre a elaboração do experimento, tarefas selecionadas ou em relação à abordagem DyeVC?