[[1]](#footnote-1)

DyeVC

Cristiano M. Cesario and Leonardo G. P. Murta

Institute of Computing

Universidade Federal Fluminense, Niteroi, Brazil

{ccesario, leomurta}@ic.uff.br

*Abstract*—Software development using distributed version control systems has become more frequent recently. Such systems bring more flexibility, but they also bring greater complexity to administer and monitor the multiple existing repositories as well as the proliferation of several branches. In this paper we propose DyeVC, an extensible tool to assist developers and repository administrators in identifying dependencies among the distributed repositories in order to help to understand what is going on around one’s repository and depict the relations between the existing repositories.

*Index Terms*—Configuration management, Context awareness, Distributed version control

# INTRODUCTION

V

ERSION Control Systems (VCS) date back to the 70s, when SCCS emerged [1]. Their primary purpose is to keep software development under control [2]. Along these almost 40 years, VCSs evolved from a centralized repository with local access, as in SCCS and RCS [3], to a client-server approach, as in CVS [4] and Subversion [5]. More recently, distributed VCSs (DVCS) arose, allowing clones of the entire repository in different locations, as in Git [6] and Mercurial [7]. According to a survey conducted among the Eclipse community [8], 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 momentum and a strong tendency in the adoption of DVCSs among the open source community.

According with Walrad and Strom [9], creating branches is essential to software development, because it enables concurrent development, allowing the maintenance of different versions of a system, the customization to different platforms and to different customers, among other features that are expected from current software systems. DVCSs include better support to work with branches [10], 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, a commit is issued based on an old revision, or updates are pushed to or pulled from other repositories. All these branches, whether explicit or not, eventually will be reintegrated to their origin by means of merge operations, reflecting the changes made.

Distributed software development, especially from the geographical perspective [11], brings a set risk factors, configuration management being one of them [12]. The increasing growth of development teams, and their distribution along distant locations – even different continents –, together combined together with the proliferation of branches, introduce additional complexity to notice actions performed in parallel by different developers. According to Perry *et al* [13], concurrent development increases the number of defects in software. Besides, da Silva *et al* [14] say that development tools control concurrent development with the assistance of VCSs, which work in majority with the concept of each developer having a private workspace. This postpones the perception of conflicts that result from changes made by co-workers. These conflicts are noticed only after a pull or a push in the context of DVCS. Moreover, Brun *et al* [15] shows 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).

By enabling repository clones, DVCSs expand the branching possibilities exposed by Appleton *et al* [16], allowing several repositories to coexist with fragments of the project history. This may lead to complex topologies where changes can be sent to or received from any repository. This scenario generates traffic similar to that of peer-to-peer applications. With this diversity of topologies, the administration of the evolution of a complex system becomes a tough task, making it difficult to find an answer to questions like: “Which clones were created from a repository?” or “What are the dependencies between different clones?”, among others.

Most of existing works deal with the last two issues, giving to the developers the perception of concurrent changes. Palantir [17], Lighthouse [14], CollabVS [18], Safe-Commit [19], Crystal [15], WeCode [20], and Polvo [21] are examples of this kind of work. Among these, the only one that deals with multiple branches is Polvo, establishing metrics that assist in determining the merge effort between branches. However, it has a strict focus in Centralized Version Control Systems (CVCS), which are much less prone to branches if compared to DVCS.

This paper expands the concept of DyeVC[[2]](#footnote-2) presented in a previous work [22], which is a novel visualization infrastructure for DVCS, which gathers information about different clones of a repository and presents them visually to the user. This allows one to perceive how his repository evolved over time and how this evolution compares to the evolution of other repositories in the project. The main goal of DyeVC is threefold: to increase the developer knowledge of what is going on around his repository and the repositories of his teammates; to enable repository administrators to visualize how the several existing repositories of a system interact with each other; and finally, to establish an extensible platform to present different information and metrics regarding distributed repositories.

The rest of this paper is organized as follows. Section II presents a motivational example to this work. Section III presents the approach used in DyeVC, along with its current state of development. Section IV shows how this work was evaluated. Section V discusses some related work and Section VI concludes the paper and presents future work.

# MOTIVATIONAL EXAMPLE

Fig. 1 shows a scenario with some developers, each one having a clone of a 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 in Xavier Institute, leaded by Professor X, and a remote developer (Storm) that periodically receives updates from the Institute. Outside the Institute, Wolverine leads a remote team located in a different continent, which is constantly synchronized with the Institute. Arrows in Fig. 1 indicate the direction in which updates are sent. Thus, for example, Rogue can pull updates from Gambit, and both Gambit and Beast can pull updates from Rogue.

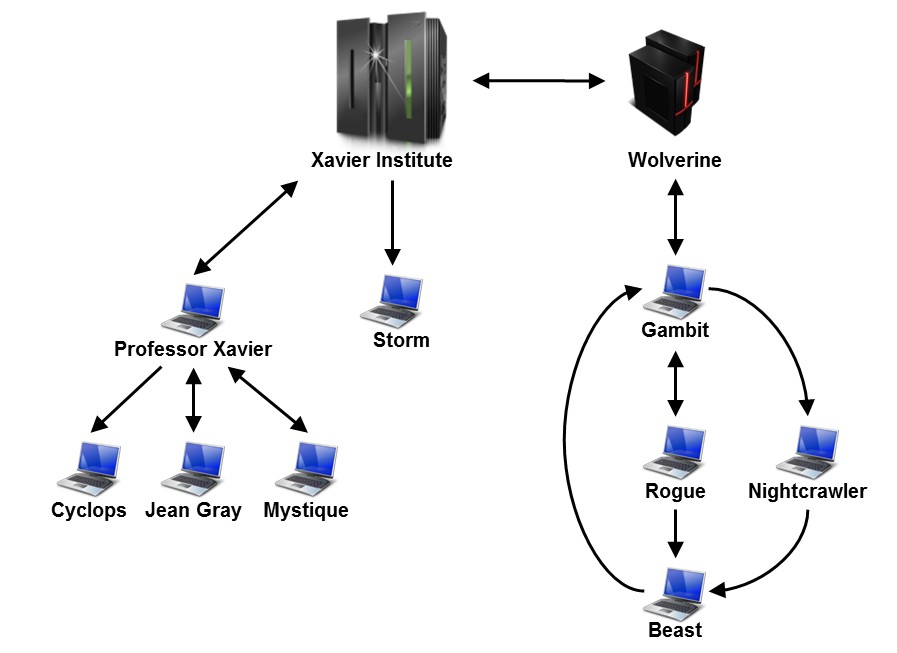


Fig. 1. A development scenario involving some developers

Each one of the developers has a complete copy of the repository and is able to send and receive updates to or from any other developer. Considering the existence of *n* developers, we could reach a total of *n \* (n - 1)* different possibilities of communication. In practice, however, this limit is not reached: while interaction among some developers is frequent, it may happen that others have no idea about the existence of some coworkers, as it occurs with Mystique and Nightcrawler, 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 or in Nightcrawler that were not yet pulled? Alternatively, would be the case that there are local commits pending to be pulled by Gambit? Beast could certainly periodically pull changes from his partners, to check if eventually there were updates, but this would be a manual procedure, prone to be forgotten. What if a tool had the knowledge of Beast’s partners, and constantly monitored those, warning Beast of any local or remote updates that had not been synchronized yet?

From an administrator’s point of view, how can he know what are the existing clones of a system? Who and how do they relate with? How can he know if there are pending commits to be sent from a test repository to a production one?

# DYEVC

The approach we propose with DyeVC involves continuously monitoring a group of interrelated repositories, starting from repositories registered by the user. The implementation uses Java Web Start[[3]](#footnote-3) Technology, and focus on monitoring Git repositories, once that this is the most used DVCS nowadays [8]. The gathering of information from repositories is accomplished using JGit[[4]](#footnote-4) library, which allows the user to use DyeVC without having a Git client installed. DyeVC presents the information gathered as a series of graphs by using JUNG[[5]](#footnote-5) library, from which it inherits the ability to extend existing layouts and filters to create new ones, which can be dynamically attached to the graphs that it presents.

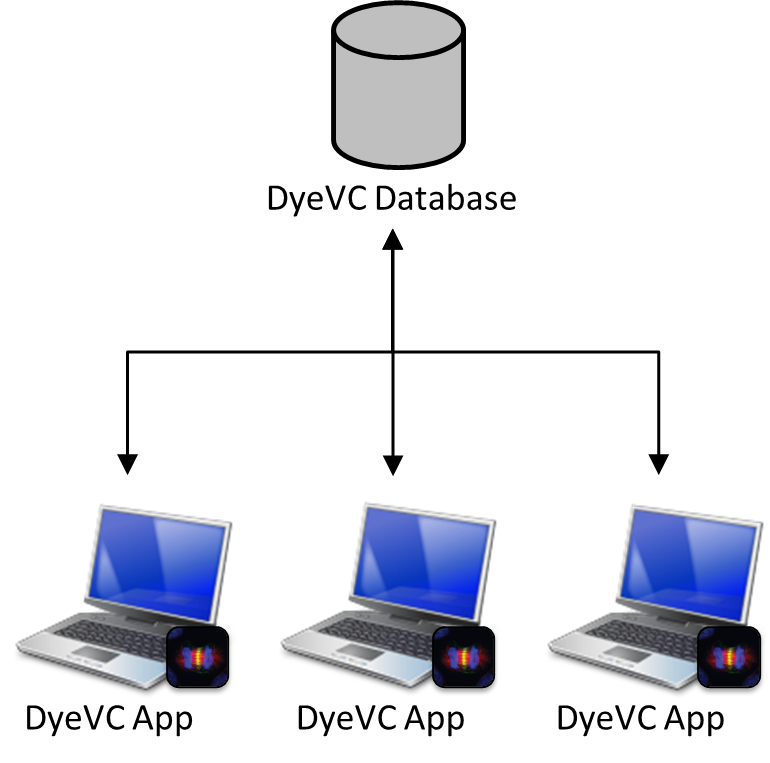


Fig. 2. How DyeVC gathers information

The information gathered is stored in a central document database running MongoDB[[6]](#footnote-6), using two types of documents:

* The repositories document, comprised of:
  + Repository id;
  + System name, used to group all repositories that are clones of a specific system;
  + Host name of the machine hosting the clone;
  + Clone name, which is given by the user when it registers the clone to be monitored;
  + Clone path in the user’s machine;
  + List of clones to which this repository pushes to;
  + List of clones from which this repository pulls from.
* The commits document, comprised of:
  + Commit id, which is the hash given by Git;
  + Commit date;
  + Committer name;
  + List of repositories where this commit exists;
  + List of this commit’s parents (considering that a commit can be a merge of several parent commits);
  + Commit message;
  + System name.

DyeVC has also the ability to gather information not only from the registered repositories in the user’s machine, but also from its partners, which are the repositories that a given repository communicates with.

## Levels of Detail

The information gathered by DyeVC is presented visually in different levels of detail. The application shows notifications whenever there are changes in any of the registered repositories or in its partners, as shown in Fig. 3. The period between subsequent monitor runs is configurable and defaults to 5 minutes.



Fig. 3. DyeVC showing notifications in the notification area

The levels of detail defined in DyeVC include presenting the status of a given repository against its partners (Level 1); zooming into the branches of the repository, showing the status of each local branch that tracks a remote branch (Level 2); and zooming into the commits of the repository, showing a visual log with information about each commit (Level 3).

Table I

Possible Status of a Repository

| Status | Description |
| --- | --- |
| question_32 | DyeVC has not analyzed the repository yet. |
| check_32 | Repository is synchronized with all partners. |
| ahead_ylw_32 | Repository has changes that were not sent yet to its partners (it is ahead its partners). |
| behind_ylw_32 | Partners have changes that were not sent yet to the repository (it is behind its partners). |
| aheadbehind_ylw_32 | Repository is both ahead and behind its partners. |
| nocheck_32 | Invalid repository. This happens when DyeVC cannot access the repository. The reason is presented to the user. |

The status evaluation considers the existing commits in each repository individually. Each commit maps a group of changes in various artifacts and it is uniquely identified in the repository. Moreover, due to the nature of DVCS, where old data is never deleted and commits are cumulative, 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. Thus, by examining the existence of commits in the local repository not yet replicated to the remote repository, and vice-versa, it is possible to come to one of the situations presented in Table II.

Table II

Status of a local repository with regard to a remote one, based on the existence of non-replicated commits in each one of them

| Existence of  non-replicated commits | | Local Status |
| --- | --- | --- |
| Local  Repository | Remote  Repository |
| Yes | Yes | aheadbehind_ylw_32Ahead and Behind (needs push and *pull*) |
| Yes | No | ahead_ylw_32Ahead (needs *push*) |
| No | Yes | behind_ylw_32Behind (needs *pull*) |
| No | No | check_32Synchronized |

To illustrate how this approach works, let us assume that each commit is represented by an integer number. At a giving moment, the local repository of some of the developers from Fig. 1 have the commits shown in Table III.

Table III

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 synchronizations presented in Fig. 1, which depend on the direction of the arrows, the perception of each developer regarding to his known partners is shown in Table IV. 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 IV

Status of each repository based on known remote repositories

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Repository | Wolverine | Gambit | Rogue | Nightcrawler | Beast |
| Wolverine | - | check_32 | - | - | - |
| Gambit | check_32 | - | aheadbehind_ylw_32 | - | ahead_ylw_32 |
| Rogue | - | aheadbehind_ylw_32 | - | - | - |
| Nightcrawler | - | ahead_ylw_32 | - | - | - |
| Beast | - | - | behind_ylw_32 | behind_ylw_32 | - |

The main window of DyeVC presents Level 1 information, as shown in Fig. 4. Upon a mouse over on a repository, DyeVC informs the number of commits that are ahead or behind in each branch that tracks a remote branch (Level 2 information).



Fig. 4. DyeVC Main Screen

Level 3 information presents a visual history of the repository (Fig. 5). Each vertex in the graph represents a known commit for the same system, which is named after its hash’s five initial characters. A hevier stroke denotes that the commit is a branch’s head (e. g. commit f1a48). Each commit is painted according to its existence in the local repository and its partners, as following:

* If commit exists locally and in all partners, it is painted in white;
* If commit exists locally but does not exist in any partner it pushes to, it is painted in green;
* If commit doesn’t exist locally, but exists in any of the partners it pulls from, it is painted in yellow;
* If commit exists in repository that is not a partner (this is, it can’t be pulled from it), it is painted in red;
* If commit does not belong to a tracked branch, it is painted in gray.

Commits are drawn according to its precedence order. Thus, if a commit N is created over a commit N – 1, then commit N will be located in the right hand side of commit N – 1. DyeVC presents a tooltip with some information about each commit, upon a mouse over a node.

The history window can also be zoomed in or out, whether the user wants to see details of a particular area of the history or an overview of the entire history. The line style can be one of cubic curves, straight or quad curves. By changing the window mode from *transforming* to *picking*, it is possible to select a group of nodes and collapse them into one node that represents them, or simply drag them into new positions to have a better understanding of an area where there are too many crossing lines.



Fig. 5. Log Window (commit history)

## Topology View

In Level 1, DyeVC presents the topology view showing all repositories of a given system, as depicted in Fig. 6, where each node represents a known clone of the system. Each node icon has a different meaning:

* A blue computer represents your clone;
* Black computers represent ordinary clones;
* Servers represent central repositories that do not pull from nor push to any other clone, or clones where DyeVC is not running;
* Green checkmarks represent selected nodes.

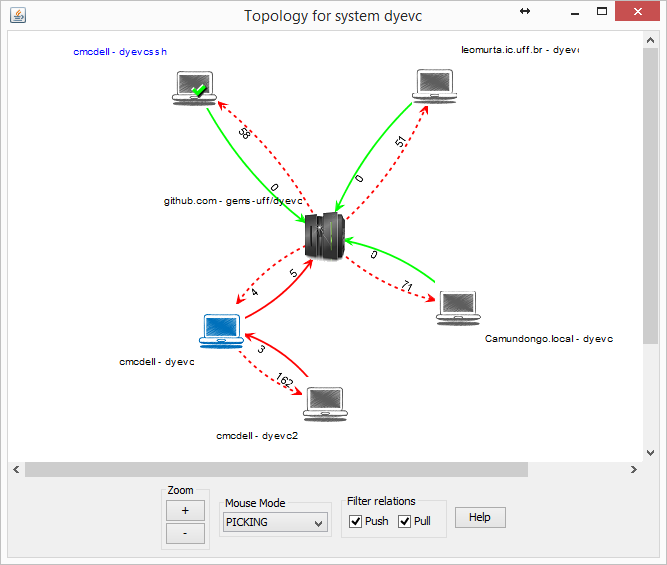


Fig. . Topology view for a given system

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 dotted stroke mean that the destination clone pulls from the source clone. The edge labels show how many commits from the source clone are missing in the destination clone. If both clones are synchronized, than the edge connecting them is green, otherwise it is red.

# Evaluation

To be written.

# Related Work

DyeVC relates primarily with studies that aim at improving the perception of developers that work with distributed software development (awareness tools). A recent work by [23] presents a systematic review of such studies. We can group these works in two categories: those that notify commit activities (conflict avoidance) and those that detect conflicts (conflict detecting).

Approaches that notify commit activities, such as *SVN Notifier*[[7]](#footnote-7), *SCM Notifier*[[8]](#footnote-8), *Commit Monitor*[[9]](#footnote-9), *SVN Radar*[[10]](#footnote-10) and *Hg Commit Monitor*[[11]](#footnote-11) focus on avoiding conflicts by increasing the developer’s perception of concurrent work. However, they fail to identify related repositories and do not provide information in different levels of details, such as status, branches, and commits. DyeVC provides these different levels of details, as shown in Section III.A.

Approaches that detect conflicts, such as *Polvo* [21] *Palantir*, [17], *CollabVS* [18], *Crystal* [15], *Safe-Commit* [19], *Lighthouse* [14], and *WeCode* [20] not only give the developer awareness of concurrent changes, but also inform if any conflicts were detected. Among these works, only *Crystal* works with DVCSs. It detects physical, syntactic, and semantic conflicts (provided that the user informs the compile and test commands), but does not deal with repositories that pull updates from more than one partner and demands having a Git client installed. On the other hand, *Polvo* presents metrics that quantify merging complexity between involving Subversion branches, which are calculated reactively (upon user request). DyeVC can be seen as a supporting infrastructure that can be combined with such approaches to allow conflicts and metrics analysis over DVCS.

# Conclusions and Future Work

Collaborative software development is a great challenge. If, on the one hand, parallelism in activities brings scale gains, on the other hand it tends to increase the concurrency, causing rework and productivity loss. The proliferation of branches and the distribution of repositories makes it difficult to realize parallel actions made by different developers. In this paper, we presented DyeVC, a tool that identifies the status of a repository in contrast with its partners, which are dynamically found in an unobtrusive way.

A number of research topics arise from this approach. The ability to discover partners of a repository induces that it is possible to find all existing clones of a repository, showing them as a network topology of interrelated nodes that communicate with each other, sending and receiving updates. A global commit history view could be drawn, showing all existing commits in all nodes in the network, allowing one to see which commits exist somewhere, and which ones have not been propagated to all nodes yet. The ability to attach new layouts and filters allows the development of new visualizations, in order to present different metrics and views of the repository (e.g. which repositories or which people changed a specific artifact or group of artifacts, which commits introduced high amount of changes in the code, which branches would cause a conflict if merged, among others).

References

[1] M. J. Rochkind, “The source code control system,” *IEEE Trans. Softw. Eng. TSE*, vol. 1, no. 4, pp. 364–470, Dec. 1975.

[2] J. Estublier, “Software configuration management: a roadmap,” in *Proceedings of the Conference on The Future of Software Engineering*, New York, NY, USA, 2000, pp. 279–289.

[3] W. Tichy, “RCS: A system for version control,” *Softw. - Pract. Exp.*, vol. 15, no. 7, pp. 637–654, 1985.

[4] P. Cederqvist, *Version Management with CVS*. Free Software Foundation, 2005.

[5] B. Collins-Sussman, B. W. Fitzpatrick, and C. M. Pilato, *Version Control with Subversion*. Stanford, CA, USA, 2011.

[6] S. Chacon, *Pro Git*, 1st ed. Berkeley, CA, USA: Apress, 2009.

[7] B. O’Sullivan, *Mercurial: The Definitive Guide*, 1st ed. O’Reilly Media, 2009.

[8] Eclipse Foundation, “The Open Source Developer Report - 2013 Eclipse Community Survey,” San Francisco, CA, USA, Jun. 2013.

[9] C. Walrad and D. Strom, “The importance of branching models in SCM,” *Computer*, vol. 35, no. 9, pp. 31 – 38, Sep. 2002.

[10] B. O’Sullivan, “Making sense of revision-control systems,” *Commun. ACM*, vol. 52, no. 9, pp. 56–62, Sep. 2009.

[11] D.-C. Gumm, “Distribution Dimensions in Software Development Projects: A Taxonomy,” *IEEE Softw.*, vol. 23, no. 5, pp. 45–51, Sep. 2006.

[12] R. D. Battin, R. Crocker, J. Kreidler, and K. Subramanian, “Leveraging resources in global software development,” *IEEE Softw.*, vol. 18, no. 2, pp. 70–77, Mar. 2001.

[13] D. E. Perry, H. P. Siy, and L. G. Votta, “Parallel changes in large scale software development: an observational case study,” in *Proceedings of the 20th International Conference on Software engineering*, Washington, DC, USA, 1998, pp. 251–260.

[14] I. A. da Silva, P. H. Chen, C. Van der Westhuizen, R. M. Ripley, and A. van der Hoek, “Lighthouse: coordination through emerging design,” in *Proceedings of the 2006 OOPSLA workshop on eclipse technology eXchange*, New York, NY, USA, 2006, pp. 11–15.

[15] Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin, “Proactive detection of collaboration conflicts,” in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, New York, NY, USA, 2011, pp. 168–178.

[16] B. Appleton, S. Berczuk, R. Cabrera, and R. Orenstein, “Streamed lines: Branching patterns for parallel software development,” in *Proceedings of the 1998 Pattern Languages of Programs Conference*, Monticello, Illinois, USA, 1998, vol. 98.

[17] A. Sarma and A. van der Hoek, “Palantir: coordinating distributed workspaces,” in *Computer Software and Applications Conference, 2002. COMPSAC 2002. Proceedings. 26th Annual International*, Oxford, United Kingdom, 2002, pp. 1093 – 1097.

[18] P. Dewan and R. Hegde, “Semi-synchronous conflict detection and resolution in asynchronous software development,” in *Proceedings of the 10th European Conference on Computer-Supported Cooperative Work*, 2007, pp. 159–178.

[19] J. Wloka, B. Ryder, F. Tip, and X. Ren, “Safe-commit analysis to facilitate team software development,” in *Proceedings of the 31st International Conference on Software Engineering*, Washington, DC, USA, 2009, pp. 507–517.

[20] M. L. Guimarães and A. R. Silva, “Improving early detection of software merge conflicts,” in *Proceedings of the 2012 International Conference on Software Engineering*, Piscataway, NJ, USA, 2012, pp. 342–352.

[21] R. Santos and L. G. P. Murta, “Evaluating the Branch Merging Effort in Version Control Systems,” in *Proceedings of the 26th Brazilian Symposium on Software Engineering (SBES)*, Natal, RN - Brazil, 2012, pp. 151–160.

[22] C. M. Cesario and L. G. P. Murta, “What is going on around my repository?,” presented at the I Brazilian Workshop on Software Visualization, Evolution and Maintenance, Brasília, Brasil, 2013, pp. 14–21.

[23] I. Steinmacher, A. Chaves, and M. Gerosa, “Awareness Support in Distributed Software Development: A Systematic Review and Mapping of the Literature,” *Comput. Support. Coop. Work CSCW*, pp. 1–46, May 2012.

1. [↑](#footnote-ref-1)
2. Dye is commonly used in cells to observe the cell division process. As an analogy, DyeVC allows developers to observe how a Version Control repository evolved over time. [↑](#footnote-ref-2)
3. http://docs.oracle.com/javase/6/docs/technotes/guides/javaws/ [↑](#footnote-ref-3)
4. http://www.eclipse.org/jgit/ [↑](#footnote-ref-4)
5. http://jung.sourceforge.net/ [↑](#footnote-ref-5)
6. https://www.mongodb.org/ [↑](#footnote-ref-6)
7. http://svnnotifier.tigris.org/ (2012) [↑](#footnote-ref-7)
8. https://github.com/pocorall/scm-notifier (2012) [↑](#footnote-ref-8)
9. http://tools.tortoisesvn.net/CommitMonitor.html (2013) [↑](#footnote-ref-9)
10. http://code.google.com/p/svnradar/ (2011) [↑](#footnote-ref-10)
11. http://www.fsmpi.uni-bayreuth.de/~dun3/hg-commit-monitor (2009) [↑](#footnote-ref-11)