VeCVL: A Visual Language for Version Control

Nathan W. Eloe, Denise M. Case
School of Computer Science and
Information Systems
Northwest Missouri State University
Maryville, MO 64468, USA
{nathane,dcase}@nwmissouri.edu

Jennifer L. Leopold
Department of Computer Science
Missouri University of Science and Technology
Rolla, MO 64468, USA
leopoldj@mst.edu

Abstract

Version control systems (VCS), such as Subversion and Git, are pervasive in industry; they are invaluable tools for collaborative development that allow software engineers to track changes, monitor issues, merge work from multiple people, and manage releases. These tools are most effective when they are a part of a developer's habitual workflow. Unfortunately, the use of these powerful tools is often taught much later in a developer's educational career than other tools like programming languages or databases. Even an experienced student's first experience with version control can be unpleasant. In this paper, the authors analyze the workflow of two common Version Control Systems with different version controls (Subversion and Git) to build a common visual language for these systems (Version Control Visual Language, or VeCVL), and show that the same visual language applies to other version control systems.

Keywords- *computer science education, education technology, pedagogy, version control, visual language*

1 Introduction

Mastery of version control is vital to a developer in modern software engineering; the scale of current projects requires collaboration from teams of developers to correctly and efficiently implement, maintain, and release software that solves real-world problems. Some projects have immense code bases, some require experts in different areas, and some (such as the Linux kernel) fall into both categories.As with any tool in a software engineer's repertoire, the VCS is most effective when used frequently; however it is one of few tools that can be detrimental to the success of a project unless its use is habitual. Poor or incorrect use of a debugger can slow down development, but a developer is unlikely to destroy a project by "jumping into" a function definition instead of "jumping over" it. Incorrectly using a source control tool can destroy other developers' changes, modify history, or make it difficult to tag and release a new version of the software. To realize the full potential of version control, the system must be used frequently, with small changes.

Unfortunately, these tools are often taught late in a student's educational career. They may be introduced in a software engineering course (often not one of the first courses a student takes), and may not be reinforced in future classes. These systems can also be difficult to use; for all their power and flexibility, they can be unfriendly and require students to make drastic changes to their normal development workflow.

This paper, in conjunction with concurrent research [1], is an investigation into embedding the use of version control within pedagogy. To simplify interaction with these tools, a visual language that supports the basic operations of common version control systems is presented. It aims to reduce the severity of the learning curve these tools have and ease the transition between different systems. Additionally, an analysis of the initial implementation of VeCVL is presented.

2 Background and Related Work

In this paper, we build on research in the areas of instructional technology and pedagogy, visual languages, and version control systems.

2.1 Instructional Technology

Software engineering and development can be challenging, and a variety of pedagogical approaches have been developed to assist with the learning process [4]. *Scaffolding* is a popular instructional technology that supports beginning students in accomplishing challenging tasks [11]. As the student becomes more proficient, the scaffolding can be

DOI reference number: 10.18293/DMS2016-027

gradually removed (through a process called *fading*) until the support is no longer needed. In this work, we propose to provide a visual language for new software engineers as a scaffolding mechanism to facilitate learning and the transition to native VCS environments.

2.2 Visual Languages

To facilitate the introduction and application of version control tools in developer's workflow, we build on the work done in the area of visual languages. A visual language is a *system of communication using visual elements rather than letter strings* [10]. The introduction of useful visual indicators has been shown to improve cognition by leveraging our human ability to see patterns and trends [5]. Our work develops icons for operations in the work flow and follows conventions for visualizing relating or overlapping features in a common way as recommended by Jones [6].

2.3 Version Control Systems

Readers interested in the history of modern version control systems (VCS) should should refer to [9]. Of interest in this paper is the development of two different paradigms of VCS: Centralized and Distributed.

2.3.1 Centralized VCS

A centralized VCS works on the principle of having a single shared repository that accepts code; multiple developers pull the latest working version from the shared repository, make changes, and then synchronize their changes to the repository [2]. This workflow is simple to understand and widely used.

One consequence of this workflow is race conditions: if two developers are working on the same code base, and the first developer pushes changes to the central repository, the second developer must first update their local copy with the changes from the global repository before they can upload their changes.

One widely adopted Centralized VCS (CVCS) is Apache Subversion, or SVN. SVN is an open source, mature CVCS that is a fully semantic successor to CVS, and is used to host code for many open source projects and groups, including WebKit and the Apache Software Foundation [2]. The basic work cycle in SVN is outlined in [8] and visualized in Figure 1.

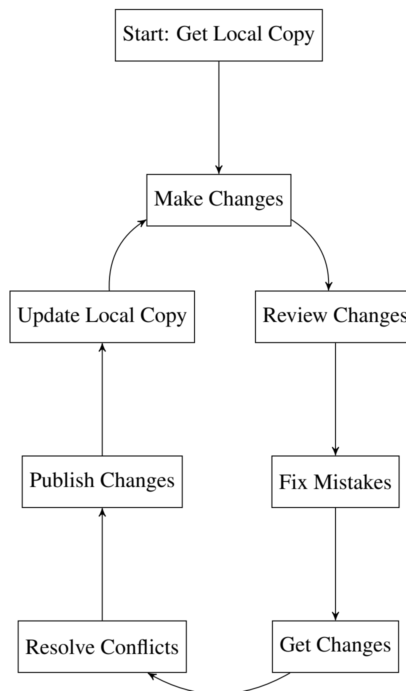


Figure 1: Basic Workflow in SVN

These basic steps map to the following commands:

- Get Local Copy: `svn checkout`
- Make Changes: `svn {add,move,copy}`
- Review Changes: `svn {status, diff}`
- Fix Mistakes: `svn revert`
- Get Changes: `svn update`
- Resolve Conflicts: `svn resolve`
- Publish Changes: `svn commit`
- Update Local Copy: `svn update`

Note that when resolving conflicts, a developer must first attempt to update their local copy with an `svn update`.

2.4 Distributed VCS

A Distributed VCS (DVCS) works on the principle that there does not need to be a central, global repository. Instead, it provides the flexibility to adopt different development styles by allowing multiple upstream repositories to be used. Git, a highly popular DVCS, is built around the concept that every copy of the repository is actually a repository that others can clone and submit changes to. This enables workflows such as the one adopted by Linus Torvalds (the creator of Git) in the development of the Linux Kernel: the Benevolent Dictator [2]. In this model, the dictator (Torvalds) maintains a *blessed repository*: a reference repository that is the "official" version. Developers update their local repositories from the blessed repository but publish changes to the dictator's trusted lieutenants. These lieutenants collect and merge these changes, providing a level of quality control and acting as an aggregation layer before code gets to the dictator. The dictator then merges changes provided by the lieutenants and publishes their master to the blessed

repository.

An interesting aspect of this style of version control is that it does not preclude the possibility of using a centralized workflow; as such, Git hosting systems like GitHub or BitBucket are frequently used as a central repository. This has one major implication: the visual language designed for a DVCS must be a superset of the visual language for a CVCS. A sample Git workflow is shown in Figure 2.

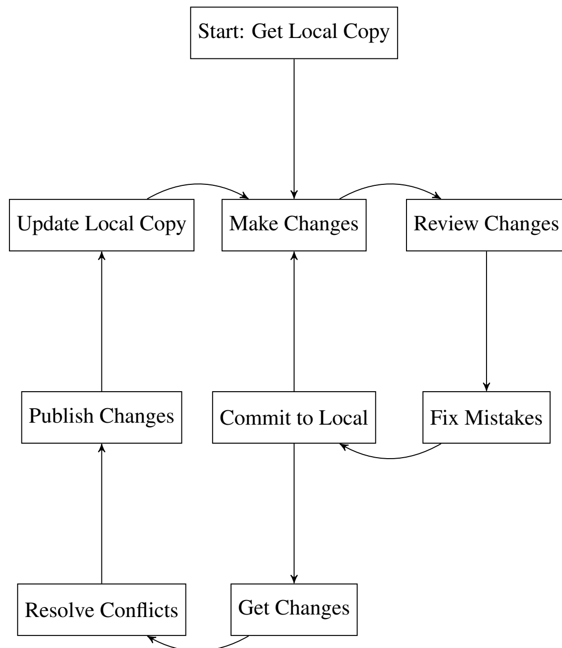


Figure 2: Basic Workflow in Git

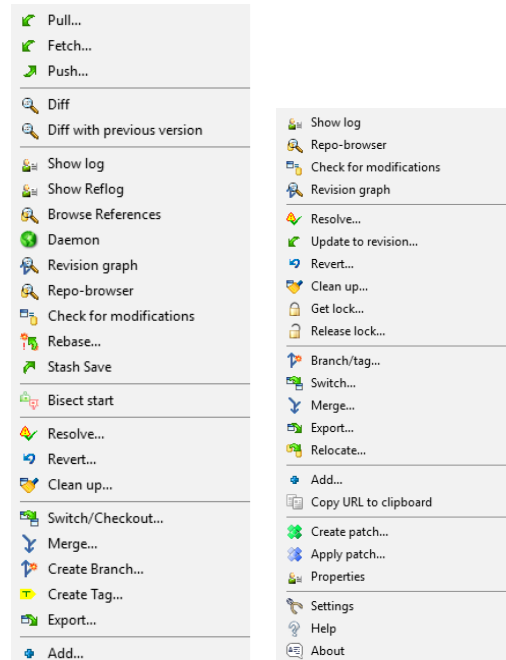
These basic steps map to the following commands:

1. Get Local Copy: `git clone`
2. Make Changes: `git {add,mv,rm}`
3. Review Changes: `git {status, diff}`
4. Fix Mistakes: `git reset`
5. Commit to Local: `git commit`
6. Get Changes: `git pull`
7. Resolve Conflicts: `git add`, `git commit`
8. Publish Changes: `git push`
9. Update Local Copy: `git pull`

Note that when resolving conflicts, a developer must first attempt to update their local copy with a `git pull`.

2.5 IDE Integration and VCS Clients

As version control is such an important tool in software development, multiple graphical interfaces for Git and SVN exist, and several IDEs have VCS Integration. Some IDEs, such as Netbeans, do not use icons in a toolbar and instead rely on drill down menus that do little to hide unneeded complexity. Figure 3 shows the icons that are used in the popular TortoiseGit and TortoiseSVN integration tools. These two programs seem to try to keep the wording and icons somewhat consistent between the two VCS.



(a) TortoiseGit context menu

(b) TortoiseSVN context menu

Figure 3: Various VCS clients and IDE VCS Integration.

3 Designing a Visual Language for VCS

3.1 Design Principles

The visual language for VCS (VeCVL) consists of two sets of icons: those for DVCS (dVeCVL) and those for CVCS (cVeCVL). VeCVL follows these design principles (listed in order of importance).

3.1.1 Centralized Version Control is a Subset of Distributed Version Control

Any icon in cVeCVL must have meaning in dVeCVL. At this point in time, VeCVL is equal to dVeCVL; however, if new paradigms of VCS are introduced, the language will be able to grow to accommodate these advances.

3.1.2 Related Operations Have Related Icons

Due to the nature of the local and remote architecture, some operations have related meaning. For example: updating the local copy makes the developer's local repository match the remote repository. Publishing modified code makes the remote repository match the developer's local repository. It follows that the icons for these operations should be similar, but in some way opposite.

3.1.3 Aggregate Operations Have Aggregate Icons

Some operations are aggregates of more primitive operations. The `git pull` command, for example, can be achieved by performing a `git fetch` (that pulls indexes of remote changes) and then running `git merge` to merge the remote changes into the local repository. If possible, the icon for `git pull` should be the composition of `git fetch` and `git merge`. This should never supersede meaning across VCS, however; if such an aggregate icon would not make sense for both SVN and Git (and other VCS), then this design principle can and should be ignored.

3.2 Verb Meanings

To design a meaningful visual language, the operations must be grouped by their basic functionality. Note that in Figures 1 and 2, the workflows are presented independently of the verb chosen by each VCS. This is important because between the two systems, the same verb may have the different meanings for Git and SVN (`commit` being the most obvious example). As such, the first step in creating a visual language is to identify an independent set of verbs that communicates the meaning of the operations. These verbs should, when possible, not be biased towards a specific VCS. Table 1 shows the identified verbs and their meanings in VeCVL. The definitions use the following glossary:

- **Upstream**: A remote repository; in a CVCS, the central repository.
- **Local**: The local working copy of an upstream repository.
- **Conflicts**: a portion of source code that has been modified both locally and upstream.

3.3 Design of Individual Elements

Because aggregate operations should have aggregate icons, there should be some commonality between elements of the language. To this end, individual parts of meanings have graphical cues that combine to form the icons.

Repositories themselves will be represented as a directory (folder); they are stored on the file system as such, so the imagery associated with it will closely mirror what it represents. Upstream repository storage will be represented by a cloud, as cloud computing has become a catch-all term to describe services and platforms running on remote hardware not under control of the user. This concept of storing shared data in the cloud should be familiar to both veteran developers and students of software engineering and provides a simple graphical element that can be used in conjunction with others to give meaning. In contrast, local storage will be represented as a computer (similar to My Computer in Microsoft Windows).

Motion between repositories or files will be represented with arrows, with the direction of the arrow indicating the target of the action. Arrows are particularly important because they are capable of representing a merge operation in addition to showing the direction of motion. The idea of a change will be represented by the Greek letter Delta (Δ), which is common notation. Other common visual cues will be drawn from established software and icons. Figure 4 shows some of the design elements that comprise VeCVL.

3.4 Resolving Conflicts In Resolving Conflicts

Git and SVN handle resolving conflicts in files in different ways. In an SVN repository, when a conflict is detected, multiple versions of the conflicted file are created. The developer then populates the file that will be kept, and runs the `svn resolve` command, indicating to the VCS that the version has the desired content.

Because Git handles the local working copy as a fully qualified repository, the normal behavior for handling merge conflicts is different. The file in question has indicators inserted that show where the conflicts occur. The developer makes changes to this file, ensures the content is correct, then simply `add`s the file to the list of changes, and makes a local commit. This `git add; git commit` workflow has the same purpose as `svn resolve`: indicate to the VCS that the contents of the conflicted file are correct. However, it treats the changes like any other modifications to the file.

Because SVN creates a new command for this operation, VeCVL will also have a verb for this action (see Table 1, under `reconcile`). When mapping to commands, it maps directly to a `git add` and a `git commit` operation. To follow the design principles specified above, the icon should be an aggregation of the icons for add and commit. This combining simple operations into more complex operations is common in Git, as it is built into the very design principles of the system.

3.5 Fetching Changes Vs. Pulling Changes

Git has an operation designed to download a listing of modifications from an upstream repository *without* merging those changes into the local repository. This `git fetch`

Table 1: VeCVL Verbs

		VeCVL Verb	VCS Verb	Meaning
dVeCVL	cVeCVL	`duplicate`	`git clone`	Create a local copy of an upstream repository
			`svn checkout`	
		`add`	`git add`	Add a file to the list of changes
			`svn add`	
		`delete`	`git rm`	Remove a file from the repository and the file system
			`svn rm,del`	
		`undo`	`git reset`	Undo changes made by a `add` operation (remove file from list of changes)
			`svn revert`	
		`move`	`git mv`	Move a file from one location to another (renames the file)
			`svn move`	
		`status`	`git status`	View status of files in repository
			`svn status`	
		`compare`	`git diff`	Compare two files, or versions of same file
			`svn diff`	
		`merge`	`git merge`	Merge two conflicting versions of code (upstream and local, or multiple branches)
			`svn merge`	
		`reconcile`	`git add,commit`	Indicate that conflicting versions of a file have been handled
			`svn resolve`	
		`sync`	`git pull`	If no conflicts, make the local repository identical to the upstream repository
			`svn update`	
		`publish`	`git push`	If no conflicts, make the upstream repository identical to the local repository
			`svn commit`	
		`commit`	`git commit`	Save a commit in the local repository (local publish)
		`fetch`	`git fetch`	Get index of changes from upstream, but do not merge into local branch

operation allows a developer to pull in changes from multiple remote systems and use changes from all of them (by treating them as separate branches of code). This does in turn mean that `git pull` is in fact an aggregate operation: `fetch` the changes from a remote repository, and then `merge` the changes from the branch representing those changes.

This is not fundamentally different from the way SVN handles the `update` operation; it simply breaks the one complex operation into two more primitive operations. The final result still involves getting the changes from the server and combining them with the local working copy. Therefore, the icon for `sync` (`git pull` and `svn update`) can be a combination of the icons for fetching and merging, without destroying the meaning of the image.

(a) Repository (b) Upstream (c) Local (d) File

Figure 4: Basic Design Elements. Icons are Modifications of [7]

4 VeCVLv1

4.1 Language Specification

Figure 5 is the result of applying the design principles and elements enumerated in the previous section. Prototype icons express the idea of the visual language. Later iterations will include increasing readability by making better use of the icon space available.

Figures 6 and 7 show the same basic workflow diagrams shown in Figures 1 and 2 respectively, but use the visual language instead of the descriptions of the steps. It is easy to see that the workflow for SVN is just a subset of the workflow for Git, which was one of the design goals for VeCVL.

4.2 Implementing VeCVL

GitSubmit [1] describes the implementation of a simplified Git client used specifically for submissions in lower level courses. It was designed to simplify the use of interacting with version control systems and get students used to the basic workflow using Git. The GitSubmit interface implements the language described by VeCVL. A thorough description of GitSubmit, including screenshots, can be found in [1].

The UI for GitSubmit is designed around the principles of design for VeCVL; indeed, the UI itself is an implementation of VeCVL. Arrows are used to denote motion of changes and commits through the system.

5 Preliminary Results

Evaluation of VeCVL as implemented in GitSubmit is ongoing through user testing. The interface is being used as the exclusive submission system in two classes at Northwest Missouri State University: Data Structures and Algorithms.

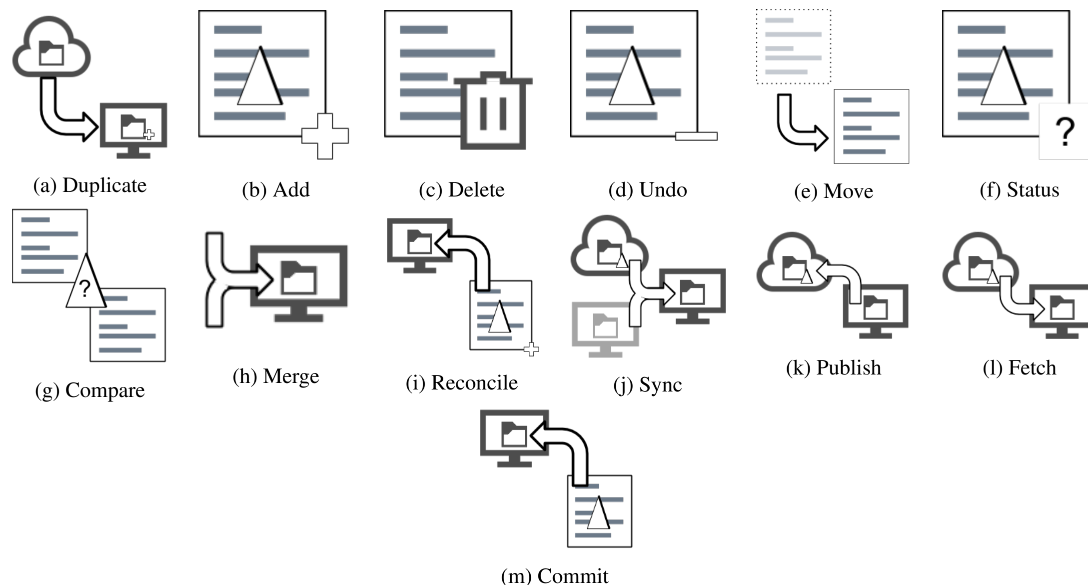

(a) Duplicate
(b) Add
(c) Delete
(d) Undo
(e) Move
(f) Status
(g) Compare
(h) Merge
(i) Reconcile
(j) Sync
(k) Publish
(l) Fetch
(m) Commit

Figure 5: Proposed Visual Language for Basic VCS Operations

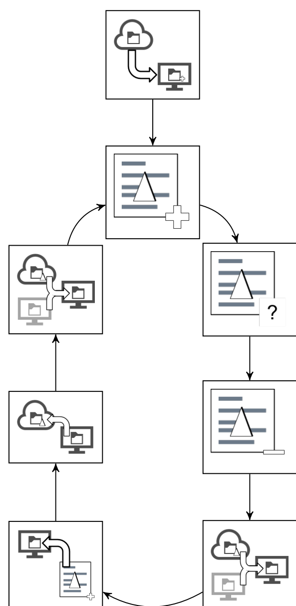


Figure 6: Basic Workflow in SVN using VeCVL

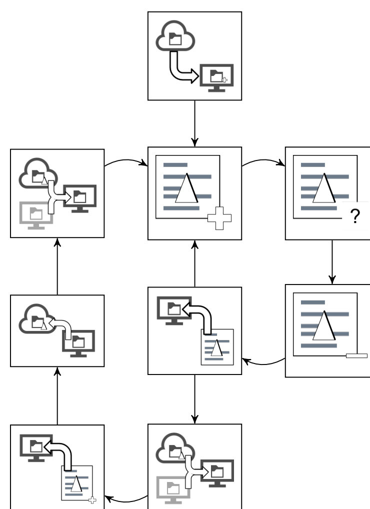


Figure 7: Basic Workflow in Git using VeCVL

Students were asked to fill out a voluntary survey about the process used in GitSubmit, and 32 responses were collected. This survey focused on certain Cognitive Dimensions of Notation. Two of the statements (with a Likert scale) directly relate to Error-proneness:

- The user interface helped avoid mistakes when submitting.
- The user interface made it easy to make mistakes when submitting.

The preliminary results for these questions is promising: 22 of 32 students either agreed or strongly agreed that the user interface helped avoid mistakes when submitting, while 8 were neutral and 2 disagreed. 20 of 32 students strongly disagreed or disagreed that the interface made it easy to make mistakes, while 6 students were neutral, and 6 students agreed with the statement. Additional feedback was collected that will be used to improve both VeCVL and GitSubmit.

6 Conclusions

Version control systems are important tools for developers working in any collaborative environment. Unfortunately, they are frequently introduced too late in a student's career for them to truly become habitual. Additionally, the complexity of interacting with these systems can scare students away from truly accepting them as a valuable part of their code development habits.

This paper introduces a visual language for version control systems as part of a multi-faceted approach to integrate version control into pedagogy. Together with [1], VeCVL aims to be a tool for teaching students how to use these tools easily. Additionally, it allows educators to teach students the *principles* of version control, not just "how to use Git" or "how to use SVN."

This initial version of VeCVL shows that it is possible to abstract away from system specific semantics and focus more on the generalized concepts behind version control. The visual language introduced in this paper covers the most basic tasks in a VCS, the ones that developers are most likely to use every day. It also follows defined design principles that should keep the language consistent and usable throughout its development. Preliminary user testing on a prototype implementation (GitSubmit) shows that the language can help with correctly using a Version Control System.

7 Future Work

This paper introduces the design and prototype implementation for a visual language for version control. The initial implementation covers basic concepts that developers use when working alone and covers the associated concepts across all of the major VCS. Future iterations are planned that will extend the visual language to include additional and powerful parts of VCS that developers use on a regular basis when working collaboratively.

Branching is a frequently used mechanism in VCS that allows multiple features and fixes to be developed concurrently without interfering with each other. The branching verb will be the next element added to this visual language. To adhere to the design principles expressed in this paper, other icons (specifically the merge/resolve action) may need to be revisited.

User testing is an important part of any visual language design. As the language is fleshed out, extensive user testing is planned that will encompass users of all levels of experience with VCS. Additional user testing will be done via surveys sent to developers in industry, academia, and students with varying levels of experience with these systems. The survey will attempt to target users of as many different version control systems as possible.

References

[1] D. M. Case, N. W. Eloe, and J. L. Leopold. Scaffolding Version Control into the Computer Science Curriculum. In *Proceedings of the 2016 International Workshop on Distance Education Technology (in conjunction with the 22nd International Conference on Distributed Multimedia Systems (DMS'16))*, 2016. In Preparation.

[2] S. Chacon. *Pro Git*. Apress, Berkely, CA, USA, 2nd edition, 2014.

[3] V. Driessen. A successful Git branching model, 5 Jan. 2010. http://nvie.com/posts/a-successful-git-branching-model.

[4] M. Guzdial. Software-realized Scaffolding to Facilitate Programming for Science Learning. *Interactive Learning Environments*, 4(1):001–044, 1994.

[5] J. Heer, M. Bostock, and V. Ogievetsky. A tour through the visualization zoo. *Commun. Acm*, 53(6):59–67, 2010.

[6] C. Jones, R. Armstrong, and K.-L. Ma. Visualizing the Commonalities Between Hierarchically Structured Data Queries. In *Proceedings of the 16th International Conference on Distributed Multimedia Systems (DMS'10)*, pages 251–256, 2010.

[7] KDE. GitHub - KDE/breeze-icons: Breeze icon theme. https://github.com/KDE/breeze-icons.

[8] M. Pilato. *Version Control With Subversion*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2004. Accessed from http://svnbook.red-bean.com/en/1.7/svn-book.pdf.

[9] N. B. Ruparelia. The History of Version Control. *ACM SIGSOFT Software Engineering Notes*, 35(1):5–9, 2010.

[10] R. Wilson and B. Levine. Medical Gas Alarm System, Oct. 29 2015. US Patent 20,150,310,718.

[11] D. Wood, J. S. Bruner, and G. Ross. The Role of Tutoring in Problem Solving. *Journal of Child Psychology and Psychiatry*, 17(2):89–100, 1976.