A Case of Computational Thinking: The Subtle Effect of Hidden Dependencies on the User Experience of Version Control

Luke Church[1], Emma Söderberg[2], and Elayabharath Elango[3]

[1] University of Cambridge, Computer Laboratory, luke@church.name
[2] Google Inc., emso@google.com
[3] Autodesk, Elayabharath.Elango@autodesk.com

Abstract. We present some work in progress based on observations of the use of version control systems in two different software development organizations. We consider the emergent user experience, and analyze the structure of the conceptual model and its presentation to see how this experience is formed. We consider its impact on the adoption of such tools outside software engineering and suggest future lines of research.

1 Introduction to Version Control Systems

Version Control Systems (VCS) are an essential part of most development workflows. They are tools that are used for managing code in different states, allowing users to access previous versions and the history of edits. They are also used to co-ordinate the development activity of multiple developers, simultaneously developing on the same code base. Commonly used examples include Subversion [10], Git [5] and Mercurial [**?**]. The use of VCSs is an important, but incidental activity. It does not directly contribute towards the creation of the software at hand. There are a number of core concepts and processes present in these systems that will be important for our discussion:

- **working copy**: the local machine has a working copy, the state of the world, that is currently accessible by software on the machine that is unaware of the version control system.
- **history**: somewhere, locally or on a separate machine, there is a history that represents alternative versions of the world.
- **change lists**: in the general operational flow, users edit their local working copy, bundle a series of edits into a change list, and then integrate it into the history. This latter step may involve resolving of conflicts due to edits by other users on the same content.

The VCS should be an ideal example of where Computer Science can help the world. The tools are undeniably useful and have been extensively tested in the field. The fundamental problem they aim to address is seen in many activities including accountancy, designing buildings and collaborative paper writing. We have observed users inventing their own ad hoc schemes for these tasks and whilst trying to solve a problem that VCSs have already solved, such as allocating unique version numbers, have ended up with data loss. In an ideal computational thinking world, a few core concepts would be widely taught, and tools used by computer scientists would be made broadly available, leveraging the complex solutions encapsulated in VCSs to improve the general population's management of information.

The desirable user experience is probably best articulated as "development without fear" [12]. That is, the ability to fluidly experiment with the confidence that you can always get back to a needed state, and to interact with others without trampling on their work. For some, this is what VCSs achieve, some interviewed users described VCSs as giving them "the freedom to experiment" and "the flexibility to try new ideas". But this is by no means the universal experience; others described the "fear of losing everything", "I dread having to use it", and talked of commands that "you'd be insane to run without copying everything first".

Other work, discussing the usability of VCSs, has concentrated on their complexity and challenges within their conceptual model [11]. We suggest that the difficult experience is not only due to lack of knowledge, but also highlight other contributing design features and consider the implications for design to support broad dissemination. We structure this discussion by first presenting two empirical studies, then consider an analytical evaluation of the user experience characteristics of an VCS, and finally give a brief design proposal and some conclusions.

2 Study Report: Google

The first and second author interviewed 5 engineers at Google, with experience of using the various VCSs at Google for between 4 months and 10 years. The engineers were engaged in various tasks ranging from managing deployment configuration systems to adding features to existing products. All of the engineers had very substantial professional software engineering experience, and used a command line-based VCS on at least a daily basis. In addition, the first author had an informal discussion with a Google engineer functioning as a team lead. During the analysis of the transcripts from the study and the informal discussion, some patterns emerged which we open coded and have separated into trends.

- **Narrow usage, except amongst team leads**: All users in the study, with the exception of the team lead, had established a workflow of a small set of commands. These workflows would typically only be altered when an unusual circumstance occurred. One user had included a command into their workflow after an incident and then stuck with that routine.
- **Risk aversion, little adoption of new features**: Three users provided examples of risk aversion in their workflow. One user had included a command in their workflow to prevent a past problem from reoccurring, with the sole purpose of preventing a problem which may not be current. Two users avoided using alternative commands that would better suit the task they were trying to perform, due to uncertainty of the consequences of running a new command.
- **Social support requirements**: With the exception of the team lead, users heavily depended on social support for learning how to use the VCS. New team members would be guided by the team lead and would learn over time with the help of peers, both locally and via peer-edited support material. Typically, these support mechanisms were used when a new unfamiliar situation occurred, often in preference to the inbuilt help mechanism of the tool. The team lead also viewed part of their responsibility as providing support for their team's use of VCSs.
- **The majority of errors are slips**: While on their commonly used path of command sequences, all users tended to only have slips. Common examples included a malformed argument, or the incorrect use of a flag.
- **Understand the basic conceptual model of used part of the VCS**: Throughout the study, all users were able to explain the basic conceptual model of how the VCS system they were using worked.
- **Startup cost**: New employees receive training on developer workflows, in addition there is self-study training material, and community support pages. These resources appear to be broadly known about and used. In addition, the knowledge transfer from engineers with more experience is important. Both of these elements can be considered startup costs for using VCSs at the company. During informal conversations about this research with another team lead, the recommendation for the team to switch VCS was accompanied with the caveat "but schedule two weeks of downtime to adapt your workflows".

3 Study Report: Autodesk

The first and third author have been engaged with a team with experience developing on Subversion that has moved to using Git. The team included a UX designer, and a number of software

engineers, test engineers, developer-relations engineers and managers. The team ranges from people with several years of experience using Git, to those new to version control. Their professional development experience ranged from 1 to 13 years. Whilst there was not a formal study on Git usage, the authors have observed a number of patterns in the team members' usage during the 9 months.

- **Narrow usage, except amongst team leads**: The usage of most of the team members other than the tech leads was restricted to a specific series of narrow, repeated interactions. When deviations away from this workflow were required to solve a problem, the team members would frequently prefer to manually 'reset' to a known state where their known workflow was functional again.
- **Risk aversion**: Proposed changes to workflows, or the use of techniques to 'repair' local states were not generally adopted. None of the team reported experimenting with features other than their established workflow, whereas during the same period they experimented with a number of different features of the programming language in use.
- **Repair operations are expensive**: Throughout the project there were a number of incidents where productivity was reduced due to issues with Git. The cost of addressing these problems was often considerable, ranging from several engineering-hours for local corruption issues to a full engineering day to remove some unwanted information from the repository. Several of the team report having to perform repeated local repairs by re-cloning their entire repository.
- **Low user confidence**: A number of the team report minimal confidence in Git. Several of the team reported manually copying of their local working set to a separate backup directory before performing operations via Git. Even one of the more experienced Git users requested that someone else perform an operation because "it scares the [elided] out of me"
- **Startup cost is perceived as being high**: New members joining the team have reported a considerable cost associated with learning how to use the source code control system. This has frequently been several days of concerted effort. Significant enough that those team members report successful basic interaction as something to mention in the daily scrum stand-up meetings.
- **Social support requirements**: During the period of observation there have been a number of occasions where the team leads have needed to assist other team members with their use of Git. In a number of cases, this occurred when the team member reported being 'blocked' by an issue with their VCS. The same has not occurred over questions about any of the features of the programming language in use.

4 Analysis - Common Themes

There are a number of elements that are common between these studies, despite being from different development organizations with different training and cultures.

Firstly, there is ritualized behaviour amongst the participants. They follow narrow workflows, changes either get integrated into that workflow and repeated each time, or they tend to 'reset' to a known workflow. These users are experienced professional abstraction workers; if anyone is qualified to manipulate the complex chains of abstract entities (multiple versions of a file, alternative worlds etc.), it would be this user population. However, the behaviour we observe is not one typically associated with mastery, it is much closer to the learned ritualistic behaviour that is sometimes found amongst novice computer users.

Furthermore, the participants in both studies *do* understand the VCSs. During the conceptual probing phase, they are able to clearly and precisely articulate the entities and relationships in their version control system. They frequently engage in social discourse around the tooling, discussing the concepts and their manipulation of them with ease. This implies that the oft asserted assumption that people engage in rituals "because they dont know any better", does

not apply here. Their conceptual understanding is strong, they are highly qualified within the domain, but this is not sufficient for them to feel confident in its behaviour, especially when the participants leave their established workflow.

A number of the participants talked about fear. They were "afraid to change their workflow", a task outside their normal workflow "scared them" and they expected very high levels of reward for change (one participant quantified it as needing a 20% productivity improvement within a week to adopt a new tool). This is the response that Attention Investment [2] would predict for a perceived high risk activity.

It is particularly problematic that a tool that would ideally provide 'development without fear' is empirically a tool whose use is perceived as being risky.

5 Analytical Evaluation

We have seen that a number of participants report that VCSs are perceived as being risky systems to use. In order to understand what might give rise to this, let us consider a Cognitive Dimensions [8] analysis of Git. The *activity* being performed is typically Incrementation (e.g. adding one more commit) or Exploratory Design (e.g. restructuring the history of the repository). There are multiple different notational possibilities, the simplest to analyze is probably the command line interface, which follows a typical noun-verb structure. e.g. `git pull` or `git branch --track MyTestBranch`. These are performed from the local working directory. Here, a few dimensions dominate:

- **Hidden Dependencies**$_{CD}$: There are many Hidden Dependencies within Git; there is a dependency that files on the disk are associated with a branch, another dependency between the local branch and the remote repository and a third that remote branched depend on other branches. Files that have been changed on disk have a dependency on their previous state (which can be accessed by performing a 'revert'). New files that have been created, but have not been explicitly '`git add`'ed, will be unknown to the version control system.
  The list of these forms of dependencies is long. They exist with regard to the branches, the commits, the stash etc. There are tools that exist to make some of these dependencies visible, but it is a challenging task to comprehend the graph of dependencies between these hidden entities. Consider the following:
  ```
  $ git status
  On branch master
  Your branch is up-to-date with 'origin/master'.
  nothing to commit, working directory clean
  ```
  This was actually in a working directory that is hundreds of changes (commits) behind the remote repository, but is up-to-date with respect to local version of the remote repository. As a further compounding factor, the Hidden Dependencies are between abstract entities.
- **Abstraction**$_{CD}$: Git has a moderate abstraction barrier. In order to start using it, the user needs to understand a few concepts (at minimum: the file, the commit, the remote repository and usually also the branch). It is not especially abstraction hungry, allowing branches to be added pretty much at the users discretion.
  Whilst most of its entities are abstract (e.g. the branch), this in-itself would be insufficient to explain the emergent properties. As noted earlier, our users were very familiar with manipulating abstract entities.
- **Premature Commitment**$_{CD}$: With a long view, VCSs can be seen as a tool to decrease premature commitment. They allow different possibilities to be explored non-destructively. However, within an episode of interaction, there is extremely high premature commitment. For instance, you must switch branches to manipulate one of the hidden dependencies before performing an operation.
- **Viscosity**$_{CD}$: The viscosity of Git is usually very low. It seldom requires repeated execution of commands to perform even very complex operations.

– **Closeness of Mapping**$_{CD}$: This will always be challenging for a VCS. Whilst the tool maps well to developer workflows, there is no physical analog for what they offer (being able to go back in time, make a different choice and merge it into the present).

None of these dimensions are themselves a direct description of the user experience, they describe the structural usability properties of the system. The experience emerges out of repeated interactions with the tool. This process of experiential emergence is discussed at considerably greater length in [3, 4].

We postulate that it is a combination of the Hidden Dependencies which leaves the user unclear of the state ('where am I? I'll just reset everything'), the within-episode Premature Commitment ('did I switch branches before doing that push? How do I back out of that?') and the issue that the dependencies are between abstract entities ('is it my local or my remote master that my branch is up to date with?') that causes the experience of fear and associated risk aversion. This analytical analysis also explains why the UI tooling for Git (Figure 1), that was used in the second study, did little to mitigate the usability difficulties. Our intent here is not to be over-critical. It is not a bad UI in itself, it provides recognition rather than recall [9], for the typical Git commands, has a clean interface with common commands clearly accessible.

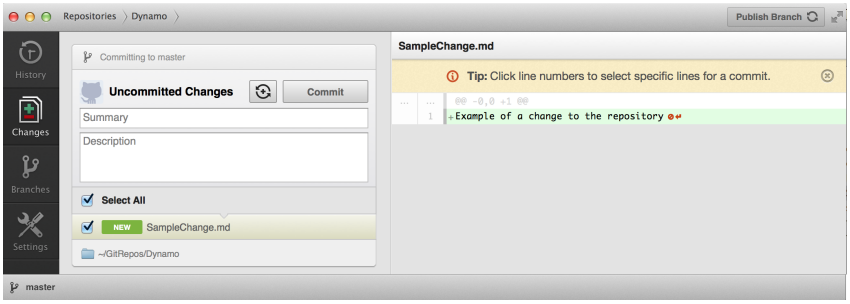

Fig. 1. GitHub's UI tool, Mac OS X, with some whitespace trimmed

However, it does little to help manage the Hidden Dependencies; the user still has to hold the state of the repository in their memory and to mentally simulate the change that is going to occur when they press commit. Or in the absence of doing so, they can just perform a ritual interaction. These dependency issues could be mitigated by making the typical design manoeuvre of making the dependencies explicit, that is, trading Hidden Dependencies for Diffuseness and depicting the states as entities that are available for more direct manipulation. Figure 2 shows examples of the possible outcomes of such a manoeuvre.

Without building it, we can of course only speculate about the usability properties such a system would have. We present the sketches here to encourages others to engage with the experiment of building a version control system that focus on the management of Hidden Dependencies and the emergent user experience of fear.

6 Implications for Design

This work has a number of implications for the design of systems, beyond the immediate example of VCSs.

1. The inclination to assert that conceptual understanding is the only missing piece of programming tooling is not empirically supported. Other aspects are crucial, especially for tools seeking broad adoption
2. Hidden Dependencies may have substantial effects on the experience of using systems, even amongst professional abstraction workers. This is especially relevant to UI libraries that are becoming increasingly dependency dense [1, 7]

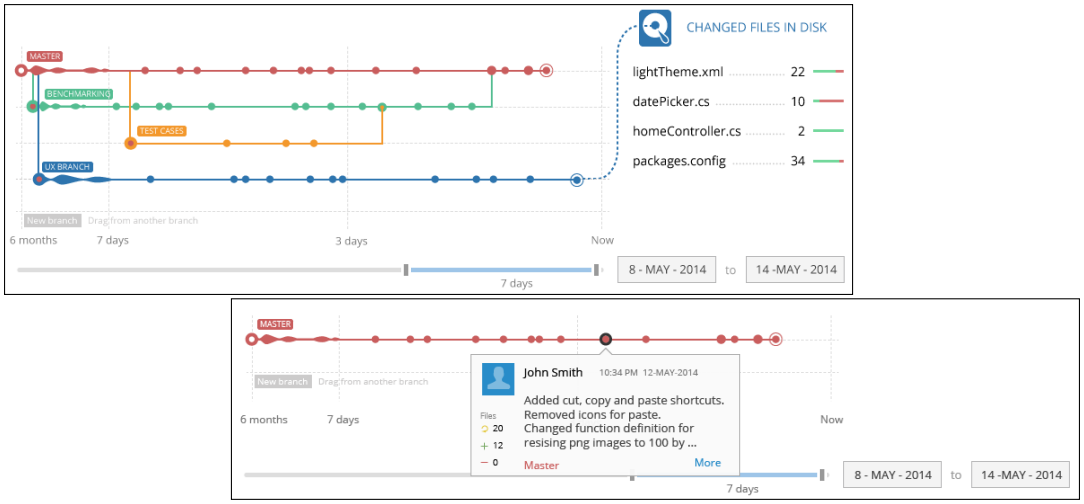

Fig. 2. Upper left: A mock of a tool that makes the changes on the brances, and the relationship between the files on disk and the files in the repository, more directly visible. Lower right: A mock for a tool providing a context plus detail view for understanding the changes that have occured on a specific branch.

3. Risk aversion amongst professionals is a potential "design smell" that indicates that a tool may have serious usability problems when adopted by a wider population

7 Conclusions

In general, VCSs provide an interesting playground for experiments in the psychology of programming. They have many of the same usability properties of programming languages, but without the extremely high implementation costs and challenges to external validity. VCSs play a central role in much software development practice, and they have much to offer to other disciplines. However, we have shown that with their current design, they have very problematic usability, even for professional software engineers. They represent an example where expertise alone is insufficient to give confidence, and have a long way to go before offering 'development without fear'. This is an example of the additional concerns beyond the strictly computational that must be addressed before the transfer of technology from Computer Science to a wider population can be successful.

References

1. Windows presentation foundation. http://msdn.microsoft.com/en-us/library/ms754130(v=vs.110).aspx.
2. Alan F. Blackwell. First steps in programming: A rationale for attention investment models. In *HCC*, pages 2–10. IEEE Computer Society, 2002.
3. Alan F. Blackwell and Sally Fincher. Pux: patterns of user experience. *Interactions*, 17(2):27–31, 2010.
4. Luke Church and Thomas R. G. Green. *A Use: Analytical methods for Usability.* 2015. To appear.
5. The Git Community. Git. http://git-scm.com/.
6. The Mercurial Community. Mercurial scm. http://mercurial.selenic.com/.
7. Google. Angularjs, 2014. https://angularjs.org/.
8. Thomas R. G. Green and Marian Petre. Usability analysis of visual programming environments: A 'cognitive dimensions' framework. *J. Vis. Lang. Comput.*, 7(2):131–174, 1996.
9. Jakob Nielsen and Rolf Molich. Heuristic evaluation of user interfaces. In Jane Carrasco Chew and John A. Whiteside, editors, *CHI*, pages 249–256. ACM, 1990.
10. The Apache Organization. Subversion. http://subversion.apache.org/.
11. Santiago Perez De Rosso and Daniel Jackson. What's wrong with git?: A conceptual design analysis. Onward! '13, pages 37–52. ACM, 2013.
12. Dagstuhl Seminar. 13382, 2013. http://www.dagstuhl.de/en/program/calendar/semhp/?semnr=13382.