Purposes, Concepts, Misfits, and a Redesign of Git

Santiago Perez De Rosso Daniel Jackson

Computer Science and Artificial Intelligence Lab
Massachusetts Institute of Technology
Cambridge, MA, USA
{sperezde, dnj}@csail.mit.edu

Abstract

Git is a widely used version control system that is powerful but complicated. Its complexity may not be an inevitable consequence of its power but rather evidence of flaws in its design. To explore this hypothesis, we analyzed the design of Git using a theory that identifies concepts, purposes, and misfits. Some well-known difficulties with Git are described, and explained as misfits in which underlying concepts fail to meet their intended purpose. Based on this analysis, we designed a reworking of Git (called Gitless) that attempts to remedy these flaws.

To correlate misfits with issues reported by users, we conducted a study of Stack Overflow questions. And to determine whether users experienced fewer complications using Gitless in place of Git, we conducted a small user study. Results suggest our approach can be profitable in identifying, analyzing, and fixing design problems.

Categories and Subject Descriptors D.2.2 [*Software Engineering*]: Design Tools and Techniques; D.2.7 [*Software Engineering*]: Distribution, Maintenance and Enhancement—Version Control

Keywords concepts; concept design; design; software design; usability; version control; Git.

1. Introduction

Experiment This paper describes an experiment in software design. We took a popular software product that is both highly regarded for its functionality, flexibility and performance, and yet is also frequently criticized for its apparent complexity, especially by less expert users.

First, we did an analysis of the product, in which we applied some new design principles [16] in an attempt to identify problematic aspects of the design, suggesting respects in which the design might be improved. Since any such analysis is likely to be influenced by subjective factors (not least our own experiences using the product, and the particular contexts in which we used it), we corroborated the analysis by examining a large number of posts in a popular Q&A forum, to determine whether the issues we identified were in fact aligned with those that troubled other users.

Second, we reworked the design to repair the deficiencies identified by our analysis, and implemented the new design. To evaluate the redesign, we conducted a user study in which users with a range of levels of expertise were asked to complete a variety of tasks using the existing and new product. We measured the time they took, and obtained feedback on their subjective perceptions.

In some respects, this project has been a fool's errand. We picked a product that was popular and widely used so as not to be investing effort in analyzing a strawman design; we thought that its popularity would mean that a larger audience would be interested in our experiment. In sharing our research with colleagues, however, we have discovered a significant polarization. Experts, who are deeply familiar with the product, have learned its many intricacies, developed complex, customized workflows, and regularly exploit its most elaborate features, are often defensive and resistant to the suggestion that the design has flaws. In contrast, less intensive users, who have given up on understanding the product, and rely on only a handful of memorized commands, are so frustrated by their experience that an analysis like ours seems to them belaboring the obvious.

Nevertheless, we hope that the reader will approach our analysis with an open mind. Although our analysis and experiment are far from perfect, we believe they contribute new ideas to an area that is important and much discussed by practitioners, but rarely studied by the research community.

Subject Git, according to its webpage, is a free and open source distributed version control system that is easy to learn, has a tiny footprint, lightning fast performance, and features

OOPSLA'16, November 2–4, 2016, Amsterdam, Netherlands
ACM. 978-1-4503-4444-9/16/11...
http://dx.doi.org/10.1145/2983990.2984018

that include cheap local branching, convenient staging areas and multiple workflows.[1] None of these claims have been disputed but one: that it is easy to learn [3, 25].

One might argue that the difficulties experienced by some users come from Git's greater power and flexibility compared to conventional, centralized version control systems. But Git, much like the Unix shell a generation ago [12], seems to elicit negative reactions from some users, who complain that Git is hard to learn and use, even for experienced programmers, and who have resorted to memorizing a short list of commands that they apply repeatedly, without fully understanding their meaning [10].

Now it is possible that these critics are to a person mistaken: that the perceived complexities of Git are inevitable consequences of its power, or are readily overcome with the right education and mindset. But it seemed to us more likely that there was something interesting going on here worth investigating, and that if we could understand what was wrong with Git—for at least a subset of its users—we might be able to extract larger lessons about software design.

Contribution The contribution of this paper is a systematic application of a theory of conceptual design to Git, which includes:

- A list of "misfit" scenarios in which Git behaves badly and flaws in concepts are to blame (§3).
- A crystallization of the purposes of version control (§4) and their connection to Git's concepts (§5).
- An application of criteria to understand why these misfits arise (§6).
- A conceptual redesign of Git (called Gitless) which attempts to fix some of these misfits (§7).
- An evaluation (§8) in two parts: a study of Stack Overflow questions (to correlate the misfit scenarios with issues reported by users); and a user study (to compare the usability and subjective perceptions of Gitless to those of Git).

This paper builds on and supersedes [22]. Compared to [22], which proposes a design for a Git reworking and includes an analysis that applies principles based on those in [8], this paper: reports on a full implementation of such a reworking; provides a much fuller analysis using the theory in [16]; and includes an evaluation in the form of a study of Stack Overflow questions and a user study.

2. Background

2.1 A Theory of Conceptual Design

The role of concepts in ensuring good design has long been recognized by researchers [7, 8, 15]. Yet most of this work focus on the design process and has little to say on concepts themselves: how they are chosen and their impact in design.

[1] Git. `http://git-scm.com`

In [16], a theory of conceptual design is proposed, which is the one we apply in this work. This section outlines the elements of the theory that are relevant to our analysis. For a detailed explanation of the theory, see [16].

The basic premise is that software design can profitably be divided into two different kinds of design: *conceptual design*, which is concerned with the selection and shaping of the essential concepts, and *representation design*, which is concerned with representing the concepts in code.

A *concept* is something you need to understand in order to use an application (and also something a developer needs to understand to work effectively with its code) and is invented to solve a particular problem, which is called the *motivating purpose*. For example, the concept of the trash in the Macintosh has the motivating purpose of enabling undoing a deletion.

A concept is defined by an *operational principle*, which is a scenario that illustrates how the concept fulfills its motivating purpose. The operational principle is thus a very partial description of the behavior associated with the concept, but focused on the particular aspect of behavior that motivated the introduction of the concept in the first place. For example, for the trash concept, the principle might state: "when a file or folder is deleted, it is not removed permanently, but saved in a special trash folder, from which it can be restored until the trash is emptied."

A concept may not be entirely fit for purpose. In that case, one or more *operational misfits* are used to explain why. The operational misfit usually does not contradict the operational principle, but presents a different scenario in which the prescribed behavior does not meet a desired goal. This goal was likely not specified explicitly; in fact, the misfit can be seen as a claim that the goals implicit in the motivating purpose should be expanded.

One operational misfit for the trash concept in the Macintosh is the following: "if the user deletes a file by mistake, and cannot remember the file's name, there is no easy way to find the file, so it may not be possible to restore it."[2] An operational misfit might be remedied by a small change in the design of the concept (for example, in this case adding the deletion date as sortable metadata), or might signal a deeper issue that is not easily resolved.

In general, the misfits of a concept design might not be found until the system is deployed and they are discovered during usage. A good design process, however, should aim to identify misfits as early as possible. One way this can be achieved is by applying application-independent criteria in the style of a heuristic evaluation [20].

The criteria we use in our analysis include:

- *Motivation*. Each concept should be motivated by at least one purpose. Concepts that are not motivated by purposes,

[2] This misfit was fixed in a recent release of OS X

but which arise, for example, from implementation concerns, usually spell trouble for a design.

- *Coherence*. Each concept should be motivated by at most one purpose. When a concept is motivated by two or more distinct purposes, it becomes hard to fulfill any one purpose well.
- *Fulfillment*. Each purpose should motivate at least one concept. If not, a purpose representing a real need will not be fulfilled.
- *Non-division*. Each purpose should motivate at most one concept. When the same purpose motivates different concepts in different contexts, confusions tend to arise about why distinct concepts are needed for the same purpose.
- *Decoupling*. Concepts should not interfere with one another's fulfillment of purpose.

2.2 Git

This section introduces the key features of Git, highlighting the more subtle ones. For a more thorough explanation, see [9, 18].

Distributed In a centralized version control system (CVCS), such as Subversion [24], there is just one repository. In a distributed version control system (DVCS), such as Git, each user has a local repository (to which work can be committed even when the user is offline). A common practice is for users to commit frequently, and only later share these changes with the rest of the team (either via "pull requests"—a request for the owner(s) of the target repository to *pull* the changes in—or by doing a *push* to the target repository).

It is common to designate a repository as a shared "remote" repository that acts as a hub for synchronizations between the "local" repositories of individual users. Typically this repository is hosted on a server and is not the local repository of any user, and is configured as a "bare" repository that has no associated working directory.

Even in this arrangement, a DVCS offers advantages over a CVCS—in particular that a user can work offline and make commits to the local repository that are pushed to the shared remote later. But Git allows more complex arrangements too, with correspondingly more complex workflows: for example, it allows a subteam to collaborate on a shared repository and then push from that repository to the repository of the larger team. This flexibility has made Git especially attractive for the development of open source projects [26].

Recording Changes Between the user's working directory and the local repository, Git interposes a "staging area." All commits are made via this intermediate area. Thus the standard workflow is first to make copies of files to the staging area (with `add`), and then to commit them to the repository (with `commit`). Explanations of Git use the term "tracked files" to refer to files that have been added to the staging area. Such files are tracked only in the sense that the `status` command will report that changes to them have not been committed. If a tracked file is updated, a subsequent `commit`—at least one without special arguments—will save the older version of the file (representing its state the last time `add` was called), and not the latest one.

Files that are currently not under version control are "untracked." Alternatively, a set of files (given implicitly by a path-specifier in a special file) may be designated as "ignored." This feature enables the user to prevent files from being committed by naming them before they even exist, and is used, for example, to prevent committing non-source files. But tracked files cannot be ignored; to ignore a tracked file one has to mark it as "assume unchanged."[3] This "assume unchanged" file will not be recognized by `add`; to make it tracked again this marking has to be removed.

A commit in Git is a complete snapshot of the repository, which is linked to its parent commits in a graph structure usually referred to as the "commit graph."

Branching A branch is a named pointer to a commit in the commit graph. Git provides the illusion that at any one time the user is working on some branch, by always updating the pointer on `commit` to point to the newly created commit. It is possible, however, to be doing work that belongs to no branch at all. Some operations can put the user in a "detached `HEAD`" state: `HEAD`, a reference that tracks where the user is working on, can "detach" from a branch and point directly to a commit, instead of pointing to one *through* a branch. New commits created in this state will not belong to any branch.

Branches are a very popular resource for keeping development tasks separate, and it is common for a repository to have many of them [2]. Switching from one branch to another enables the user to put aside one development task and work on a different one (e.g., to pursue the implementation of a feature, or fix a bug [23]). But switching branches can be a complex matter, because, although branches are maintained separately in the repository, there is only one working directory and one staging area. As a result, when a user switches branches, files may be unexpectedly overwritten. Git fails with an error if there are any conflicting changes, effectively preventing the user from switching in this case. To mitigate this problem, Git provides a way to save versions of files to another storage area, called the "stash," using a special command issued prior to the branch switch.

Tagging A tag is a named pointer to a commit. Unlike a branch, however, a tag does not get spontaneously updated on `commit`. Tags are used to note that a certain commit is special (for example, that it corresponds to a new release).

Integrating Changes Git provides three ways of integrating changes: *merge*, *rebase* and *cherry-pick*. The *merge* of two (or more) branches results in a new "merge" commit

[3] Assume unchanged was intended to be used as a performance optimization but has since been appropriated by users as a way to ignore tracked files. The current advice is to use the "skip worktree" marking instead

with two (or more) parents. The *rebase* operation changes the "base" of the current branch to the other branch: it looks for the closest common ancestor (the point at which the branches diverged), and applies all commits from the current branch that follow the common ancestor placing them *after* the commits of the other branch. The `rebase` command also has an interactive feature that allows the user to customize how commits should be applied; for example, it is possible to squash multiple commits into one, or split a commit into multiple commits. Finally, *cherry-pick* allows the user to specify an arbitrary set of commits to be applied to the current branch.

If conflicts occur the user has to fix them and let Git know by *staging* each conflicted file before being able to continue the operation.

Syncing with Other Repositories Crucial to the understanding of how syncing with other repositories work is the notion of a "remote branch," also called a "remote-tracking branch." This is a branch (pointer to a commit) that (asynchronously) reflects the state of a branch in another repository. It is updated whenever there is some network communication (e.g., a `push` or `fetch`).

The notion of a "remote branch" must not be confused with that of an "upstream branch." An upstream branch is a convenience for users: after the user assigns it to some branch, commands like `pull` and `push` default to use that branch for fetching and pushing changes if no branch is given as input.

3. Operational Misfits

In this section, we outline the key "operational misfits" of Git: scenarios in which Git behaves in a way that is unpredictable or inconvenient. These misfits are focused on fundamental aspects of the behavior, and not on quirks of the command line interface, which are a different (and more readily addressed) concern. These misfits emerged from a number of sources: critical mentions of Git in online forums and blogs, discussions with colleagues, and our own experience. Their identification preceded the analysis of Stack Overflow (§8.1) that we conducted later.

We don't expect every Git user reading this paper to resonate with these misfits, since one's perception of a misfit is likely to depend on personal experience and context of use. While the choice of misfits is inevitably subjective, what we are trying to show is that there are at least objective ways to structure and talk about design. We hope expert users will put aside the cognitive bias that arises from the "curse of knowledge" when reading this section.

Saving Changes Suppose you are in the middle of a long task and want to save your changes, so that they can be later retrieved in case of failure. How would you do that? You can create a new commit, but if the changes don't (yet) constitute a logical group you'll have to amend the commit afterwards. Alternatively, you could copy the files out to some other location and use that as a backup. But both of these options demand extra work. Many users use a cloud storage service such as Dropbox[4] for this purpose, but this comes with its share of problems (such as corrupted repositories or loss of data).

Switching Branches Suppose you are working on one branch and want to switch to a different branch (e.g., to fix a bug). Additionally suppose that the uncommitted changes you've made so far in the current branch conflict with changes in the destination branch. Switching branches is then a complex task, because you can't switch without overwriting your changes. To work around this problem you can create a new commit with unfinished work, but you'll have to amend it afterwards so that it reflects a logical group of changes. Alternatively, you can use stashing to save your changes, but when using multiple branches it becomes difficult to remember and apply the stash that corresponds to the branch you want. And even if you overcome that problem, if you are in the middle of a merge, rebase, or cherry-pick, stashing won't help since conflicting files cannot be stashed.

Detached Head Suppose you are working on some branch and realize that the last few commits you did are wrong, so you decide to go back to an old commit to start over again. You checkout that old commit and keep working creating commits. You might be surprised to discover that these new commits you've been working on belong to no branch at all. To avoid losing them you need to create a new branch or reset an existing one to point to the last commit.

File Rename Suppose you rename a file and make some changes to it. If you changed a significant portion of the file, then, as far as Git is concerned, you didn't rename the file, but it is instead as if you deleted the old file and created a new one (which means that the file history is now lost). To work around this, you have to be diligent about creating a commit with the rename only, and only then creating a new commit with the modifications. This, however, likely creates a bogus commit that doesn't correspond to a logical group of changes.

File Tracking Suppose you create a new file and then you add the file to start tracking changes to it. You keep working on the file making new modifications and then you make a vanilla `commit`. You might be surprised to find out that what actually got committed is the old version of the file (representing its state the last time the file was staged), and not the most recent one.

Untracking File Suppose there's a database configuration file committed in the repository and you now want to edit this file to do some local testing. This new version of the file should not be committed. You could always leave out the file from the commit every time, but this is laborious and

[4] Dropbox. `http://dropbox.com`

error-prone. You might think that you could make it ignored by modifying the `.gitignore` file but this doesn't work for committed files. The way to ignore this file is to mark it as "assume unchanged," but this marking will be cleared when you switch to another branch.

Empty Directory Suppose you are starting a new project and have decided on a directory structure and now want to share it with your team. Unfortunately, empty directories can't be committed. The workaround is to add a token file in each directory, which will allow you to create a commit with your directory structure, and then push it so that your collaborators can see it. Only after there's at least one file in each directory can you remove the token file.

4. Purposes for Version Control

An analysis of any system or application's fitness for purpose must begin by articulating the purpose that it is intended to fulfill. This turns out to be much harder than one might imagine, even for systems that perform familiar tasks such as version control. Partly this is due to the inherent difficulty of precisely capturing an intuitive purpose: what exactly is version control? And partly this is due to the power of tools such as Git, which fulfill, at least in part, purposes (such as backup) that are secondary to the primary purposes for which they were designed.

In this section, we attempt to capture the key purposes that are associated by users with modern VCSs in general. These are developed in a top-down manner (based on requirements). At this point, our concern is not to characterize Git (which we attempt to do later in §5) but rather to establish a kind of benchmark for a generic VCS against which Git can be evaluated.

The purposes are classified into the following categories: *data management*, *change management*, *collaboration*, *parallel development*, and *disconnected operation*.

Data Management Data management corresponds to the idea of a "backup," namely the ability to recover data if is lost (either by corruption or accidental deletion). Most backup systems also have a way of recovering *old* versions of the data (and not just the last one saved). This is not "version control," but is a fundamental feature of backup, since it mitigates the risk of changes that are both intended and integrity preserving, but are regretted subsequently.

Purpose 1. Make a set of changes persistent
Save changes to data so that they can be later retrieved in case of failure

Change Management Change management distinguishes VCSs from backup systems. A VCS should provide not only the means to save and restore versions of files (data management), but also the means to manage *change*.

Purpose 2. Group logically related changes
Group a set of changes together, to make it possible to return to a set of versions of files that are in an appropriate relationship to one another (for example, reflecting new functionality that crosses multiple files), and to allow changes to many files to be handled in aggregate

Purpose 3. Record coherent points
Amongst several groups of changes, mark some as significant, because they correspond to coherent points in the development

What constitutes a coherent repository depends on the team. It will typically include issuing a release, but may also include points at which the code is in a suitable state for subsequent development.

Collaboration To this point, all the purposes might apply in the context of a single user. Collaboration needs arise when multiple users work together on a single code base.

Purpose 4. Synchronize changes of collaborators
This involves identifying conflicts, and providing means for their resolution

Parallel Development Parallel development corresponds to the ability to maintain multiple parallel and independent lines of development. Typical uses include: maintaining multiple releases of the same software, keeping the development of major features isolated, and trying out experimental changes without affecting the main line [1, 23, 28].

Purpose 5. Support parallel development
Switch between, and synchronize changes of, different lines of development—even when the work on a particular line is incomplete

Disconnected Operation Disconnected operation refers not only to the ability to work without an Internet connection but also to the ability to disconnect from your collaborators: to work while ignoring changes made by others until they prove necessary. A crucial difference between CVCSs and DVCSs is how well they fulfill this purpose. In a CVCS, you get a (local) private working directory, thus you are able to modify files without having to worry about changes being made by other collaborators. But as soon as you want to make a commit, you have to sync with the changes made by others. On the other hand, in a DVCS you can delay connecting (to the rest of your collaborators or to the Internet) until you want to share your changes.

Purpose 6. Do work in disconnected mode
Obtain the benefits of version control (as described in the other purposes) without requiring an Internet connection and coordination with other collaborators

5. Purposes and Concepts of Git

In this section, we attempt to outline the concepts of Git, and to identify the purposes that they address. This is an essential step in evaluating Git, since it allows the elements of the design to be evaluated independently of one another. Without it, evaluating fitness for purpose would be a largely unstructured activity.

A good concept should have a compelling purpose: that purpose is what allows users to grasp the concept, and allows developers to focus on designing the concept to fulfill that purpose. In contrast, the conventional view is that the concepts of an application fulfill one or more purposes only in aggregate; there need be no rationale for the design of any single concept except that it plays some role in the larger whole. Our view is that this amorphous relationship between concepts and purposes is what has hindered the kind of design analysis we are attempting here, and that the approach of assigning purposes to concepts not only immediately highlights some discrepancies, but also provides a decomposition that makes deeper analysis possible. In particular, when concepts seem to have no direct mapping to a given purpose, their motivation is questionable, and one might wonder whether they are needed at all.

We therefore do not expect the designers and users of Git to share exactly our classification of purposes. On the other hand, the very lack of consensus found today makes this a productive exercise. If the outcome of this work were a more general agreement about which essential purposes Git is intended to fulfill, and which are incidental and secondary, that would, in our view, represent concrete progress.

Table 1 shows the *motivating purposes* for key concepts of Git. We also relate each motivating purpose to the set of purposes for generic version control (using the $\mapsto$ symbol to stand for "is a subpurpose of"). The motivating purposes are reverse engineered from our reading of popular Git references [9, 18] and from our own experience with Git. Due to space constraints we don't include operational principles.

5.1 Stashing: An Example

We consider the motivating purpose of stashing not to be a subpurpose of any of the high-level purposes for version control (§4). This section elaborates on our rationale for this.

Take (what seem to be) the motivating use cases for stashing [9, Chapter 7.3]: (1) to pull into a dirty working directory and (2) to deal with an interruption in your workflow.

The first refers to the case where you have some uncommitted changes that happen to conflict with the changes being pulled. The workaround is to create a stash (which will clean the uncommitted changes), do the pull (which will now succeed since there are no conflicting changes), reapply the stash to your working directory, and fix conflicts.

The second refers to the case where your uncommitted changes prevent you from being able to do some unrelated work. You might be working on some feature on the main branch and now need to work on some other feature instead (and you want to put the changes made so far temporarily aside), or maybe you have to switch to a different branch but are unable to do so because your uncommitted changes conflict with the changes in the destination branch.

In both cases, what makes stashing compelling is the fact that it is a quick and easy way of *cleaning* the working directory (thus leaving you in a state where it is possible to do your next task) *while saving* the uncommitted changes.

Two candidate high-level purposes are to make a set of changes persistent (P1) and support parallel development (P5). But the connection to both of these is tenuous. For the former, while stashing might be seen as a way to persist a set of changes this doesn't account for the cleaning of the uncommitted changes, a crucial motivation for stashing. For the latter, while one could argue that stashing is letting you switch (and later synchronize) between two (implicit) lines of development (one that has your uncommitted changes, and the other one that consists of the new changes to be pulled or developed), these are hardly genuine lines of development, but are ad hoc states of branches.

This lack of connection doesn't mean stashing isn't useful in Git. On the contrary, not having stashing would make a lot of tasks in Git significantly harder. Stashing does have a single, coherent motivating purpose. The problem is the lack of connection between this purpose and the high-level purposes for version control, which suggests that the introduction of stashing might be to patch flaws in the design of Git and not to satisfy a requirement of version control.

6. Analysis

Now we apply the design criteria (§2.1) to Git's purposes and concepts (§4, §5) in an attempt to understand why the misfits (§3) arise. Table 2 summarizes our results.

Incoherent Commit The problem with commit is that it constitutes a violation of the coherence criterion: the same concept (commit) has more than one, unrelated, purpose: make a set of changes persistent (P1) and group logically related changes (P2).

These two purposes are not only unrelated, but in tension with each other. On the one hand, you would like to save your changes as often as possible, so that if something bad happens you lose as little data as possible (thus encouraging early committing). On the other hand, a logically related group of changes usually involves multiple individual changes, which means that you might be working for quite some time before you have enough changes to group (thus encouraging late committing).

As misfit "saving changes" illustrates, this forces users to either commit often and then fix the commits so that they constitute a logically related group of changes (say, using the interactive features of `rebase`), or resort to doing some out-of-band action to backup changes.

Concept	Motivating Purpose
Repositories	
Non-bare Repository	Do work in disconnected mode (P6)
Bare Repository	Serve as a hub for users to upload/download changes $\mapsto$ synchronize changes of collaborators (P4)
Commit	Make set of changes persistent (P1) and group logically related changes (P2)
Working Directory	Create, read, update and delete files $\mapsto$ make a set of changes persistent (P1)
Staging Area	Select and review the changes that will go in the next commit $\mapsto$ group logically related changes (P2)
Stash	Clean up uncommitted changes, while saving them so that they can be later reapplied $\mapsto$ no purpose (see §5.1)
References	
Head	Name the commit you are currently basing your work on $\mapsto$ make a set of changes persistent (P1)
Tag	Record coherent points (P3)
Branch	Support parallel development (P5)
Upstream Branch	Set a default for sync operations $\mapsto$ synchronize changes of collaborators (P4)
Remote Branch	Do work in disconnected mode (P6)
File Classifications	
Tracked	Group files that are currently under version control $\mapsto$ group logically related changes (P2)
Untracked	Group files that are currently not under version control or ignored $\mapsto$ group logically related changes (P2)
Ignored	Prevent committing file $\mapsto$ group logically related changes (P2)
Assume Unchanged	Prevent committing file $\mapsto$ group logically related changes (P2)
In Conflict	Mark files with conflicts that need to be resolved manually $\mapsto$ synchronize changes of collaborators (P4)

Table 1: Key concepts and purposes of Git

Concept/Purpose	Violates Criterion	Causes Misfit
Commit	Coherence	Saving Changes
Branch	Decoupling	Switching Branches, Detached Head
Stash	Motivation	Switching Branches
File Classifications	Decoupling	File Tracking
Assume Unchanged, Ignored	Non-division	Untracking File
Saving renames	Fulfillment	File Rename
Saving directories	Fulfillment	Empty Directory

Table 2: Analysis summary

Branch Coupling The misfit "switching branches" is caused by the violation of the decoupling criterion by the working directory, staging area and branch: the working directory and staging area interfere with the fulfillment of the motivating purpose of the concept of a branch, i.e., to support parallel development (P5).

In addition to this, there is the coupling that causes misfit "detached head." Since the head can detach (from a branch) and point directly to a commit, the user can inadvertently create a new line of development that is not referenced by any branch. The problem here is that the head is interfering with the purpose of branching due to having the potential of creating a line that is hard to switch back to.

Unmotivated Stash The problem with the concept of stash is that it's unmotivated. In §5.1 we argued why its purpose doesn't map to any high-level purpose. We believe the reason for this is that the motivating purpose for the inclusion of stash results from other decisions in Git's design (e.g., how branching works), and not from the intrinsic complexity of version control.

File Classifications Coupling The problem with the staging area is that it seems to be always in the middle, occasionally causing unexpected behavior such as the one illustrated by misfit "file tracking." This misfit is caused by the violation of the decoupling criterion by the staging area and file classifications. Contrary to what a novice user might expect, a tracked file is not automatically considered for commit; it needs to be staged first (either explicitly via `add` or implicitly by passing some flag to `commit`).

Divided Ignored and Assumed Unchanged The problem with assume unchanged is that it violates the non-division criteria: the same purpose "prevent committing file" motivates both the concept of an ignored file and assume unchanged file. So the user needs to learn another concept and separate set of commands to do the same thing that the ignore mechanism provides, causing misfit "untracking file."

Unfulfilled Purposes Misfits "file rename" and "empty directory" reflect a violation of the fulfillment criterion, since they correspond to purposes that are subpurposes of "make a set of changes persistent" (P1).

7. Gitless

7.1 Overview

Gitless has no staging area, and the only file classifications are "tracked," "untracked," "ignored," and "in conflict." A tracked file is a file whose changes will be detected by Gitless (and automatically considered for commit); an untracked file is one whose changes will not be detected by Gitless; ignored and in conflict are the same as in Git. An important distinction is that files can move freely between these classifications; it does not matter whether the file has a version at the current commit point (head) or not.

A branch in Gitless is a completely independent line of development: each branch includes the working version of files (i.e., it is as if there is a working directory per branch), maintains the information about file classifications (i.e., a file could be tracked in some branch but untracked in another), and also maintains the information of any sync operation in progress (i.e., even during merges with conflicts, the user is still working on the original branch, and can switch to another branch and then go back to fixing conflicts later). Also, there is no possible way of getting in a detached head state; at any time, the user is always working on some branch (the "current" branch). Head is a per-branch reference to the last commit of the branch.

Finally, as in Git, Gitless has bare and non-bare repositories, a working directory, and tags.

7.2 Purposes and Concepts of Gitless

The changes made to Git's concept model are:

1. The redefinition of "tracked" and "untracked," and the elimination of "assume unchanged" and the "staging area."
2. The redefinition of the concept of "branch," and the elimination of "stash."
3. The creation of the notion of a "current branch," and the redefinition of "head."

Change 1 prevents misfits "file tracking" and "untracking file;" change 2 prevents misfit "switching branches;" and change 3 prevents misfit "detached head." The remaining misfits are not addressed by Gitless.

7.2.1 Discussion

Our redefinition of the file classifications makes them easy to control. As a result, there is no need for an "assume unchanged" classification, since tracked files can be easily ignored (or made untracked).

Gitless eliminates the concept of a file having a staged version, and there is a single and direct path (in both directions) between working and committed versions.

Common use cases for the staging area in Git are to select files to commit, split up a large change into multiple commits, and review the changes selected to be committed. We address the first by providing a more flexible `commit` command that lets the user easily customize the set of files to commit (with `only`, `include` and `exclude` flags). For the second use case we have a `partial` flag in `commit` that allows the user to interactively select segments of files to commit (like Git's `commit --patch`). Finally, our `diff` command accepts the same `only`, `include` and `exclude` flags to customize the set of files to be diffed. There could be other use cases for the staging area that Gitless doesn't handle well but we expect these to be fairly infrequent.

In regard to branching, to address situations in which the user wants changes made in the current branch to be moved onto the destination branch (e.g., the user realizes that she has been working in the wrong branch), the Gitless `branch` command has a `move-over` flag.

More importantly, if the user has any uncommitted changes and attempts to do any synchronization operation (that conflict with the uncommitted changes) the uncommitted changes in the working directory are saved and later reapplied after the operation finishes.

Our concept of branch (and the mechanics of the synchronization operations) eliminate the need for stashing, a concept that is not present in Gitless. The main uses cases for stashing (pulling into a dirty working directory, and dealing with an interrupted workflow) can be now easily fulfilled without it.

Details of Gitless commands are given on the website[5].

8. Evaluation

For the evaluation, we sought to answer two questions: (1) To what extent do the misfits identified illustrate real problems Git users face? (§8.1) and (2) To what extent does Gitless address the misfits identified in Git and what's the resulting impact on usability? (§8.2)

8.1 Stack Overflow

In an attempt to determine how closely the problems we identified match problems users experience in practice, we performed a manual analysis of Stack Overflow posts on July 18, 2016. We started by finding all questions with more than 30 upvotes tagged with the keyword "git." We then examined, by hand, the resulting questions (roughly 2400) to determine whether or not the issue being asked about is related to one of the misfits. We considered a question to be related only if there's evidence that the original poster (OP) is experiencing the complication illustrated by the misfit. If it's hard to tell, based on the information available in the question, we don't include the question in our results. Take, for example, the question "Squash my last X commits together using Git."[6] This question might be related to misfit "saving changes" since the OP might have done bogus commits to achieve persistence and now wants to group changes. But it may also be the case that the OP's commits represent logical groups of changes that now need to be regrouped. Since it's hard to tell, we don't include the question in our results.

Despite our analysis being conservative, we found 41 highly ranked questions that match the misfits. The question titles, with their respective upvote and view counts are shown, clustered by misfits, in Table 3. The sheer number of aggregated views per misfit suggests that these misfits represent real problems experienced by Git users.

For misfit "saving changes," most questions (Q1-4, Q6-7) are about trying to fulfill the purpose of making a set of changes persistent ("backup"). One is about trying to fix an unfortunate situation caused by using Dropbox and Git together (Q5).

[5] `http://gitless.com`

[6] `http://stackoverflow.com/questions/5189560`

Misfit		Question	Upvotes	Views
Saving Changes	Q1	Using Git and Dropbox together effectively?	927	215523
	Q2	Backup a Local Git Repository	122	78674
	Q3	Fully backup a git repo?	54	37502
	Q4	Is it possible to push a git stash to a remote repository?	105	30820
	Q5	Git fatal: Reference has invalid format: refs/heads/master	90	25717
	Q6	Is "git push –mirror" sufficient for backing up my repository?	34	18415
	Q7	How to back up private branches in git	33	10580
Switching Branches	Q8	The following untracked working tree files would be overwritten by checkout	365	378331
	Q9	git: Switch branch and ignore any changes without committing	148	129120
	Q10	Why git keeps showing my changes when I switch branches (modified, added, deleted files) no matter if I run git add or not?	47	10524
Detached Head	Q11	Git: How can I reconcile detached HEAD with master/origin?	784	397694
	Q12	Fix a Git detached head?	490	397985
	Q13	Checkout GIT tag	125	98328
	Q14	git push says everything up-to-date even though I have local changes	113	79203
	Q15	Why did my Git repo enter a detached HEAD state?	202	78856
	Q16	Why did git set us on (no branch)?	65	41866
	Q17	gitx How do I get my 'Detached HEAD' commits back into master	136	42794
File Rename	Q18	Handling file renames in git	315	242864
	Q19	Is it possible to move/rename files in git and maintain their history?	367	153701
	Q20	Why might git log not show history for a moved file, and what can I do about it?	34	17099
	Q21	How to REALLY show logs of renamed files with git?	60	12923
File Tracking	Q22	Why does git commit not save my changes?	177	142189
	Q23	Git commit all files using single command	165	141815
Untracking File	Q24	Ignore files that have already been committed to a Git repository	1588	387112
	Q25	Stop tracking and ignore changes to a file in Git	975	353136
	Q26	Making git "forget" about a file that was tracked but is now in .gitignore	1458	286435
	Q27	git ignore files only locally	562	120700
	Q28	Untrack files from git	218	140663
	Q29	Git: How to remove file from index without deleting files from any repository	110	61498
	Q30	Ignore modified (but not committed) files in git?	135	38293
	Q31	Ignoring an already checked-in directory's contents?	169	49692
	Q32	Apply git .gitignore rules to an existing repository [duplicate]	40	28286
	Q33	undo git update-index –assume-unchanged <file>	165	37262
	Q34	using gitignore to ignore (but not delete) files	55	23381
	Q35	How do you make Git ignore files without using .gitignore?	58	23709
	Q36	Can I get a list of files marked –assume-unchanged?	191	20184
	Q37	Keep file in a Git repo, but don't track changes	74	15572
	Q38	Committing Machine Specific Configuration Files	58	5934
Empty Directory	Q39	How can I add an empty directory to a Git repository?	2383	432218
	Q40	What are the differences between .gitignore and .gitkeep?	841	121484
	Q41	How to .gitignore all files/folder in a folder, but not the folder itself? [duplicate]	227	80119

Table 3: List of misfits with their related Stack Overflow questions

Some users expect branching to work just as in Gitless. In Q10, the OP states, "I thought that, while using branches, whatever you do in one branch, it's invisible to all the other branches. Is not that the reason of creating branches?" In Q8-9, the OP is trying to switch branches, but uncommitted changes prevent her from doing so.

For "detached head," all questions (Q11-17) are of users that inadvertently got their repository into a detached head state, are confused about it, and now need help to get their repository back to a sane state.

Questions for "file rename" (Q18-21) all arise from cases in which users are trying to figure out how to get Git to, as Q21 says, "really" track renames.

In Q22 the OP is confused about the staging area and wondering why `commit` doesn't save the changes. In Q23, the OP wants to simply skip it altogether.

There's a myriad of questions about how to untrack a committed file (Q24-32, Q34-35, Q37-38). Those who figured out that the way to do it is by marking the file as assumed unchanged are left wondering how to list this kind of file (Q36) or how to undo the marking (Q33).

The need for sharing empty directories is so common (Q39, Q41) that there's a convention to use the name `.gitkeep` for the bogus file added to an empty directory, making novices wonder what the difference is between `.gitkeep` and `.gitignore` (Q40).

8.2 User Study

Through the week of August 24-28, 2015, we conducted a usability test in which we recruited Git users and ask them to complete a series of short tasks using Git and Gitless (a so-called "within-subjects design"). The goal of the study was to evaluate the usability impact of the conceptual transformations applied in Gitless to address misfits.

Participants were recruited by an email sent through a public lab mailing list composed mostly of current and alumni students, faculty, and research staff. In the study application we asked applicants to rate their own proficiency using Git ("novice," "regular user," or "expert user"), indicate how often they used Git ("several times a day," "several times a week," or "several times a month") and what they used Git for. Since the experiment was to be conducted using a Unix shell we also asked them if they were comfortable using one and whether they have used Gitless before. We also ask them to report their occupation in general terms (e.g., student, software engineer).

The study consisted, for each participant, in two sessions of (roughly) 1-hour duration each. Participants completed one session using Git and the other using Gitless. To account for learning effects, participants were randomly assigned to use Git or Gitless for their first session, and the sessions were scheduled at least a day apart from each other. In each session, participants found themselves in an imaginary scenario where they were part of a team developing a software artifact. The team used Git(less) to track changes to code. We didn't require participants to know how to code; instructions were provided on what to change. What was left for them to figure out was how to use Git(less) to complete the tasks.

Participants were allowed to use the command line help (e.g., `git help`, `git submcd -h`, `gl subcmd -h`) as well as any resource on the web. Note that while there's a wealth of information about Git on the web, for Gitless the only resource is the webpage manual[7]. A 3-minute overview of Gitless was given to participants before their Gitless session.

The tasks were performed on a desktop machine with Mac OS v10.10, Git v2.5 and Gitless v0.8.2. We used QuickTime to record the screen. (No audio was recorded.) A slide presentation was used to present the tasks. Each task (or subtask for the long tasks) was described in one slide. We asked participants to walk through the slides and complete the tasks in order. They could take breaks between tasks.

After each session, we asked them to complete a short questionnaire about their experience using Git(less) to complete the tasks. We collected data using 7-point Likert scales on satisfaction, efficiency, difficulty, frustration and confusion. At the end of their second session, we asked them to complete an additional final questionnaire comparing Git and Gitless with respect to learnability and easiness of use, rate how much they enjoyed using Gitless, and state whether they would continue using it if they could.

8.2.1 Tasks

Each session had a total of 6 tasks (plus one practice task). Each task had a number of steps to be completed in order. To provide guidance, each step was usually accompanied with a set of command executions and their expected output. These were intentionally vague so that participants could take different paths to reach the same end state.

In each session, participants where asked to role-play as a member of the team of "Fit Industries Inc.," developers of "fit-cli." Fit-cli is a command line tool (written in Python) that given your gender, age, weight and height, computes an estimate of the number of calories you burn per day just for being alive (also known as the resting metabolic rate).

Table 4 includes a description of the tasks we asked participants to do and their relation to the misfits (§3). Task 2 was a buffer task to build an interesting repository and leave the participant with uncommitted changes that are relevant for Task 3. For the full list of steps per task, including the changes to files they were asked to make, see the appendix.

The tasks were specifically designed to put participants in challenging situations when using Git. For example, in Task 1, to create a commit that includes all changes made to the file, the participant needs to remember to stage the file again before commit. In Task 3, uncommitted changes prevent a clean switch, to fix the bug participants need to create a stash or an intermediate commit. At the beginning of Task 4 there are two stashes in the stash stack: the top one is the one corresponding to the meters feature and the other one to the kilos feature (the right one to apply for that task). An execution of `git stash pop` with no argument would therefore apply the wrong stash. In Task 5, since stashing doesn't work while in the middle of conflicts, participants need to find a workaround. One option is to copy the changes made so far out of the repository, abort the rebase, and when back at fixing conflicts retrieve the changes. Others involve continuing the rebase with an intermediate commit and later amending it. Finally, in Task 6, if the participant were to checkout the previous commit this would put the user in a detached head state, which causes problems later.

8.2.2 Measures

- *Task success rate*: percentage of participants that completed the task successfully. We say a participant completed the task successfully if the state of the repository at the end of the task matches the expected one. Minor differences in the content of files or in commit messages are ignored.
- *Task completion time*: time taken to complete or abandon a task.

8.2.3 Participants

We recruited 11 participants (9 male, 2 females), ages 20 to 38 (M=26.09). They consisted of students (3 undergraduates, 2 graduates), researchers (3) and software engineers (3). None of them classified their Git proficiency as "novice," 10 "regular" and 1 "expert." 7 typically used Git "several times a day," 3 "several times a week," 1 "several times a month." None of the subjects had used Gitless before. All reported being comfortable using a Unix shell. Six participants used Git for their first session; the rest used Gitless. For the time and trouble of participating in the study, a $30 gift card was given to participants.

8.2.4 Results and Discussion

To establish an objective classification that would allow us to better separate questionnaire results by Git proficiency

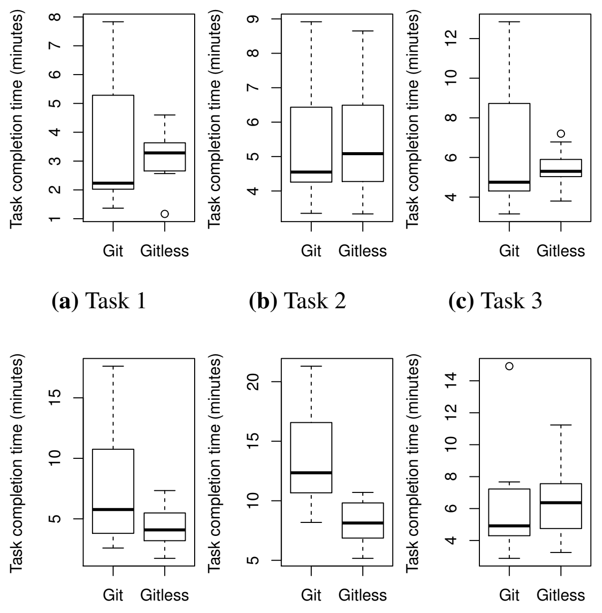
[7] `http://gitless.com`

Task	Description	Misfit
1. Add readme file	Create a new file (`readme`), track it, make another modification to it, and create a commit that includes all changes made to the file	File tracking
2. Let users input weight in kilos	Create a new branch `feat/kilos`, switch to it, make a change and commit. We then ask them to make another change that is left uncommitted	No related misfit (see §8.2.1)
3. Let users input height in meters	Create a new branch `feat/meters`, switch to it and make a change. The participant then needs to switch to `master` to fix a bug	Switching branches
4. Wrap with features	Go back to working on the kilos feature, which involves switching to `feat/kilos` branch and bringing back uncommitted changes	Switching branches
5. Fixing conflicts	Switch to another branch in the middle of conflicts	Switching branches
6. Code cleanup	Undo an unpushed commit (as if it never existed before)	Detached Head

Table 4: List of tasks with their related misfit

Task	Success Rate		
	Git	Gitless	Difference
1. Add readme file	81.82%	100.00%	18.18%
2. Let users input weight in kilos	90.91%	63.64%	−27.27%
3. Let users input height in meters	72.73%	81.82%	9.09%
4. Wrap with features	54.55%	63.64%	9.09%
5. Fixing conflicts	54.55%	90.91%	36.36%
6. Code cleanup	63.63%	81.82%	18.90%

Table 5: Task success rates



(a) Task 1 **(b)** Task 2 **(c)** Task 3

(d) Task 4 **(e)** Task 5 **(f)** Task 6

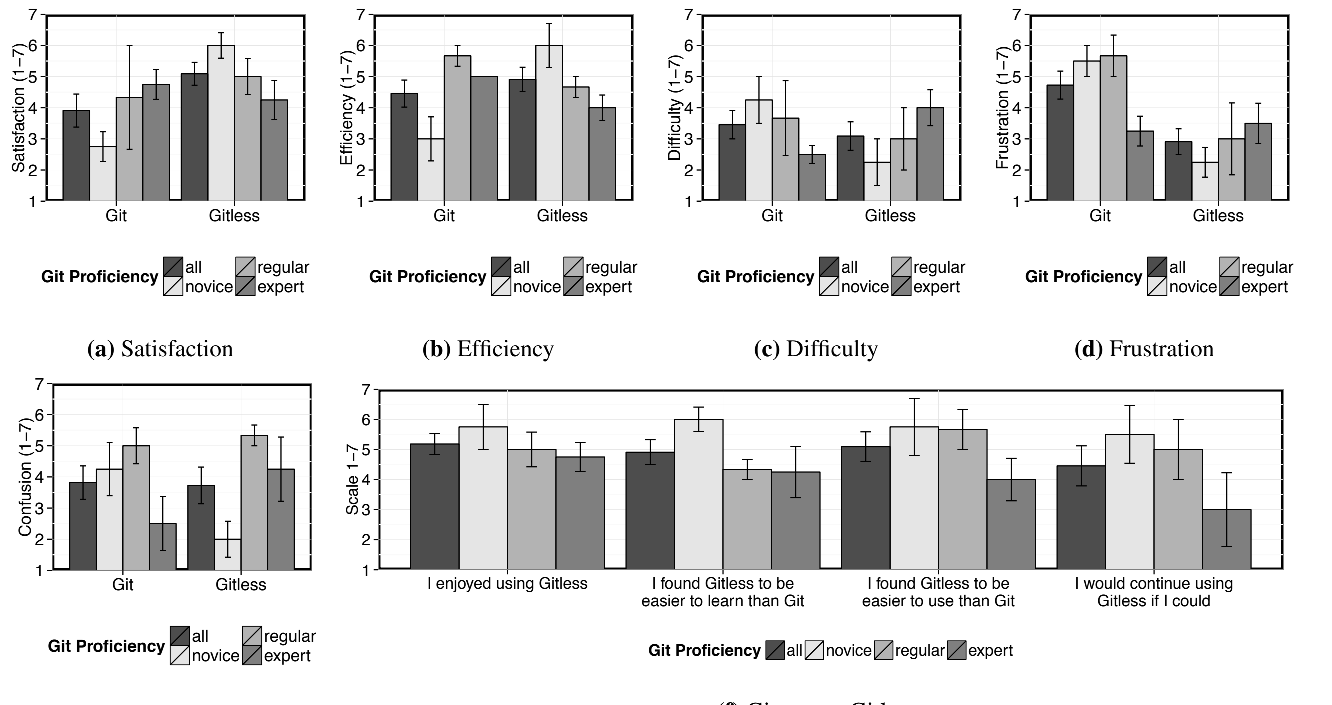
Figure 1: Box plots of task completion times

we ran a k-means clustering algorithm (k=3) using Git task completion times and got a total of 4 novices, 3 regular and 4 experts. We had planned to rely on our subjects' own classifications of their level of Git proficiency, but we found this to be more subjective than we had anticipated, with their perceptions influenced by how they use Git and how aware they are of what they do not know. We therefore decided instead to establish a more objective classification by clustering them based on Git task completion times.

Task Success Rates Task success rates are shown in Table 5. Overall, participants did better using Gitless. In Task 2, participants that failed the task using Gitless (4) did so because they never switched to branch `feat/kilos` after creating it. We think the reason for this could be a confusion with `gl branch -c` where most assumed that it would not only create the branch given as input but also switch to it (like `git checkout -b`) when in fact it does not.

Task Completion Times Task completion times are shown in Fig. 1. There is more variance in the completion time for Git. This is perhaps because of the different proficiency levels participants had with Git, which caused them to struggle with tasks in varying degrees. None of the participants had used Gitless before, and the 3-minute overview created a uniform understanding, so they all spent a similar amount of time doing the tasks. Most participants completed tasks 3, 4, and 5 (which are all branching related) faster when using Gitless than when using Git. A paired t-test found the differences in task 5 to be significant (t=3.95, df=10, p=0.003). For task 4 the result was p=0.066. In all of these tasks, having truly independent lines of development proved useful. Some participants (1 novice, 1 regular, 1 expert) highlighted branching in Gitless: *"Branch handling was way more intuitive than with git. I would use gitless to deal with branches"*, *"Keeping branches separate is great. [...] Transitions between branches are very smooth"*, *"I really enjoyed the fact that one can transition between branches without committing or staging—that's a killer feature."*

Questionnaire Results Questionnaire results are shown in Fig. 2. Overall, participants found Gitless more satisfying than Git (M_{git}=3.91, M_{gl}=5.09) and less frustrating (M_{git}=4.73, M_{gl}=2.91) but there's no big difference in efficiency (M_{git}=4.54, M_{gl}=4.91), difficulty (M_{git}=3.45, M_{gl}=3.09) and confusion (M_{git}=3.82, M_{gl}=3.72). This apparent contradiction might be due to the fact that all of the participants had used Git before but were encountering Gitless for the first time without any substantive training. Some participants (2 regular, 1 expert) commented that indeed their problems with Gitless were mostly due to their lack of practice using it: *"The hardest part was learning the new commands. With more experience, I can see how this could be a better way of using git"*, *"Overall, the frustrations I ran*



(a) Satisfaction **(b)** Efficiency **(c)** Difficulty **(d)** Frustration

(e) Confusion **(f)** Git versus Gitless

Figure 2: Post-session and post-study questionnaire results (1=strongly disagree, 4=neutral, 7=strongly agree), with standard errors bars

into with gitless were because I wasn't familiar enough yet to know the terms/commands, while my frustrations with git were due to a limitation of the tool", *"Most of what slowed me down was still thinking in git commands rather than gitless commands. [...] I have over 6 years of experience with git and less than an hour with gitless."*

A paired t-test found the difference in satisfaction for novices significant (t=-3.81, df=3, p=0.032). (p=0.134 for all proficiency levels.) We also found the difference in frustration for all proficiency levels and for novices significant (t=2.60, df=10, p=0.026 and t=3.81, df=3, p=0.032).

Results comparing Git with Gitless are encouraging. Novices specially liked it, while experts didn't find it worse than Git. Overall, participants enjoyed using Gitless (M=5.18) and found it easier to learn (M=4.91) and use (M=5.09). One (novice) participant stated *"I found myself using* `status` *and* `diff` *less often because the simplified workflow and terminology gave me greater confidence that my mental model matched Gitless's."* When asked if they would continue using Gitless the results are somewhat split (M=4.45). Some (1 regular, 1 expert), for example, showed concern about its power: *"Gitless was easier to use for the tasks these sessions asked me to perform, but I really like having a Git stash and staging area to work with in Git"*, *"[...] the ability to walk away from a branch in any state is very useful and would go far in helping new git users [...] However, I make heavy use of the staging area and interactive rebase and I would not be willing to part with either."* These comments are not surprising since Gitless is a mere prototype while Git has been in use for over 10 years. (Also, at the time of the experiment, we didn't have a `partial` flag to select segments of files to commit, or a command to cleanup history.)

Note that while results suggest that our redesign efforts were fruitful (especially for novices, without a notable negative impact on experts) this doesn't mean Gitless is a "better" VCS than Git. Our study focused only on misfits and did so in a controlled environment. A full evaluation of a VCS would require testing it in the context of large projects with complex requirements. Yet our results provide some empirical evidence that suggests our approach can be profitable in addressing design-related usability problems.

8.3 Threats to Validity

Internal In addition to the conceptual model, the type (e.g., command language, direct manipulation), and quality of the user interface affects usability. This is not a major factor in our study, since Gitless has a command line interface that follows the same Unix conventions as Git; the only differences are in the command names (and of course their semantics).

External The user study was conducted on only 11 people that are, or have previously been, affiliated with computer science at MIT and may not generalize to Git users in general. To mitigate this factor Gitless is available online for free, and anyone can download and try the tool. Our findings may not generalize to real-world usage if the tasks that occur in the wild are significantly different from the ones in our experiment. The fact that we have been using Gitless for over a year now and haven't encountered any big limitations mitigates this concern. But there may be Git usage patterns that are not well supported by Gitless, especially those that use features such as stashing for special purposes.

Reliability The auxiliary material that accompanies this paper includes the Stack Overflow questions we analyzed and all necessary resources to replicate the user study. A detailed description of the tasks is included in the appendix.

9. Related Work

Our focus here is not on the theory of concept design we apply in this study; see [16] for that.

Version Control Most work on version control focus on its usage [13, 17], branching practices [1, 2, 5, 23, 28], impact in software development [4, 6, 26, 27], and on the transition from CVCSs to DVCSs [11, 19]. Our work instead looks at the problem of version control from a design perspective. The closest related work is [10], where the authors apply the cognitive dimensions framework [14] to Git.

Attempts to Fix Git Gitless is certainly not alone in its attempt to remove some of Git's complexity. Other popular attempts include GitHub's desktop client[8], legit[9], and Easy-Git[10]. These tend to focus mostly on changes in the user interface presentation, through more consistent terminology in commands and documentation, and graphical user interfaces. Gitless is a deeper reworking, and appears to a user more like a different VCS than the same VCS with a different user interface. Recent releases of the GitHub desktop app are becoming more like our redesign however, as they introduce new concepts (such as "pull requests"); the GitHub client is becoming more of a GUI for GitHub than for Git.

Gitless and Other VCSs There's a myriad of VCSs out there. Many are centralized and thus fundamentally different from Gitless (and Git). Among the DVCSs, Mercurial[11] [21] is the most popular DVCS after Git and often regarded as easier to use than Git. It is therefore worth discussing the relationship between Mercurial and Gitless in more detail.

Mercurial has no staging area, which greatly reduces the complexity related to recording changes and brings it closer to Gitless's design. The file classifications are not as controllable as in Gitless though. For example, it is possible to change the status of a file from tracked to untracked but only if the file was not committed before.

A branch is a named, linear sequence of changesets. The user is always working on some branch, so problems like being in a detached state cannot happen in Mercurial. But branches don't keep the working directory information separate, so switching branches, as in Git, can be a complex task with uncommitted changes. In addition to the concept of "branch," Mercurial also has a "bookmark." These are pointers to commits, similar to Git's branches. While a changeset records the branch in which it was made, it doesn't record bookmarks. But these don't help either.

Mercurial has a concept of tag as well, which is the same as Gitless's (and Git's). But Mercurial also has a special "floating" tag named "tip" that identifies the newest revision in the repository (serves the role of "head"). Interestingly, there is a file `.hgtags` that tracks the current tags of the repository. When a new tag is created the `.hgtags` file is automatically modified for the user and a new commit is created. This is different from how Gitless (and Git) handle tags. We believe Git's way of handling tags (which we inherit in Gitless) is better since it keeps tags and commits independent from each other.

10. Conclusion

This work applies a theory of conceptual design [16] to Git. We identify "misfit" scenarios where Git behaves badly and concepts are to blame. We perform a manual analysis of Stack Overflow questions and present evidence that suggests misfits correspond to real problems Git users' face. We propose Gitless, a new VCS that aims to fix many of the misfits identified by reworking Git's conceptual model so that it better aligns with criteria proposed in [16]. To evaluate Gitless, and see how effective is in fixing misfits, we conducted a small usability test in which we gave participants a set of tasks to complete using Git and Gitless. The results from our experiment suggest our approach can be profitable in identifying, analyzing and fixing usability problems. We don't claim Gitless is novel by itself; it's the approach that delivered Gitless that we believe to be the primary contribution of this work.

There are two ways to look at this paper. On one hand, is an attempt to improve Git and VCSs in general. While version control is an active research topic (e.g., see [5, 6, 13, 19], few look at the problem from a design perspective ([10] is one example). VCS design is an understudied topic and yet it can have a significant impact in software development. We hope to help by stimulating discussion on purposes and concepts of version control.

On the other hand, this work is a systematic application of a design theory to a widely used application. Through it we hope to contribute to the study of concept design, show its value in practice, and spur interest in viewing software this way. We show how abstract ideas like purposes and concepts can be applied to a real software system. Design research like this is messy: instead of being a hard, easily quantifiable topic, it relies more on compelling and well-argued ideas than we are perhaps used to. But if we want to make progress on software design, this might well be the only way to do so.

[8] `https://desktop.github.com/`

[9] `http://www.git-legit.org/`

[10] `https://people.gnome.org/~newren/eg/`

[11] `https://www.mercurial-scm.org`

Acknowledgments

Thank you to the anonymous paper and artifact reviewers for their useful feedback. Thank you also to the users of Gitless for their enthusiasm, bug reports, and feature requests. This research is part of a collaboration between MIT and SUTD (the Singapore University of Technology and Design), and is funded by a grant from SUTD's International Design Center.

A. Appendix

A.1 User Study Tasks and Analysis

For the interested reader, here we include a detailed description and analysis of the tasks given to participants of the user study. Due to space constraints we don't include the set of check commands. Task 0 (Practice Task) asked them to clone the fit-cli repository, which left them ready to start making changes to files.

A.1.1 Task 1: Add Readme File

Table 6 shows Task 1, Figure 3 shows the modification steps they had to perform as part of the task.

Only one participant forgot to stage the new changes to the file (step 1.3) before doing the commit (step 1.4). Despite the difference showing up in the check commands, the participant moved on to the next task instead of fixing the incorrect commit. Another participant did a commit right after doing an `add` of the file (i.e., after step 1.2). It is unclear why the participant did this, perhaps it is because of the confusion around what constitutes a tracked file (some explanations of Git talk about tracked files being those files that are committed in the local repository). The participant didn't amend the commit later to match the history. The remaining 9 participants completed the task successfully (81.82% task success rate).

We were expecting more people to end with a commit that doesn't include all changes. But in hindsight, all participants use Git often, and commit is a very common task to perform, so chances are they already have had the experience of dealing with the subtleties that arise from having an explicit staging area interposed between the working directory and local repository (and thus unlikely to miss the fact that a second `add` was required).

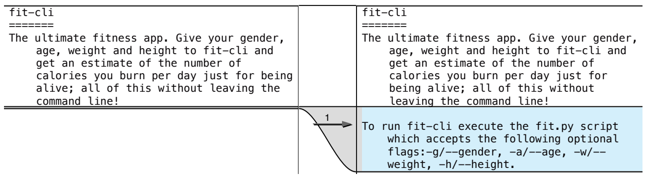
Of the 9 participants that completed the task successfully, 6 did an explicit `add` before doing the `commit` for step 1.4, the other 3 used the `a/all` flag of `commit`.

When using Gitless all participants completed the task successfully.

Task 1: Add readme file
Maybe you didn't notice but our app has no readme file! What an atrocity! Let's fix this.

1. Create `README.md` with these contents
2. Track `README.md`
3. Let's add usage instructions to the readme. Open `README.md` and make this change
4. Create a commit with all changes to `README.md` and message "added readme"

Table 6: "Add readme file" task (Task 1)



```
fit-cli
=======
The ultimate fitness app. Give your gender,
    age, weight and height to fit-cli and
    get an estimate of the number of
    calories you burn per day just for being
    alive; all of this without leaving the
    command line!
```

```
fit-cli
=======
The ultimate fitness app. Give your gender,
    age, weight and height to fit-cli and
    get an estimate of the number of
    calories you burn per day just for
    being alive; all of this without
    leaving the command line!

To run fit-cli execute the fit.py script
    which accepts the following optional
    flags:-g/--gender, -a/--age, -w/--
    weight, -h/--height.
```

(a) Step 1.3

Figure 3: Modification steps for "Add readme file" task (Task 1)

A.1.2 Task 2: Let Users Input Weights in Kilos

Table 7 shows Task 2, Figure 4 shows the modification steps they had to perform as part of the task.

Task 2 is a buffer task so as to build an interesting repository and leave the participant with uncommitted changes (step 2.5) that will be relevant in Task 3. All but one of the participants completed the task successfully (90.9% success rate). The one that didn't, skipped step 2.1 (i.e., did all the work—steps 2.2 through 2.5—on the `master` branch). For step 2.1, of the participants that completed the task successfully (10), 7 of them did `git checkout -b feat/kilos` to both create and switch to the branch in one command (the remaining 3 did `git branch feat/kilos` followed by `git checkout feat/kilos`).

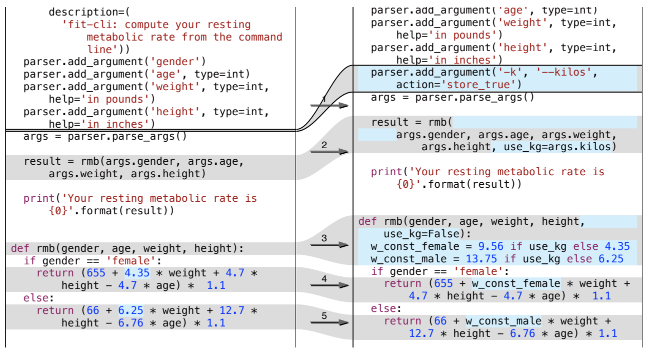
The biggest surprise was that using Gitless participants did much worse. Four participants, created branch `feat/kilos` (step 2.1) but never switched to it, and did all the work on the `master` branch instead (63.64% task success rate). Perhaps

Task 2: Let users input weight in kilos
The fact that users can only input their weight in pounds and height in inches happens to be a huge deterrent to non-US users of fit-cli. Alice (the CEO) has instructed you to implement two new features: let users input weight in kilos and let them input height in meters.
First, we are going to begin by implementing the kilos feature.

1. Let's be diligent and work in a separate branch. Create a new branch `feat/kilos` and switch to it
2. Open file `fit.py` and make this change
3. Create a commit with the changes to `fit.py` and message "kilos feature"
4. Bob (the CTO) will review our changes. Run the `ut-pr-kilos-send` command to send a pull request
5. Bob has sent you some comments on your code. First, he suggested we add a help message to the kilos flag. That's an easy fix! Open `fit.py` and make this change

Bob also suggested that instead of using the `w_const_female` and `w_const_male` variables we should change that to a dictionary. This is going to take more time to fix, so we are going to put this aside for a while and work on the meters feature instead.

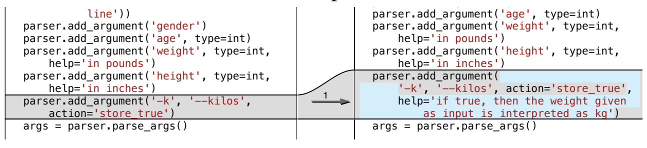
Table 7: "Let users input weights in kilos" task (Task 2)



```
    description=(
      'fit-cli: compute your resting
          metabolic rate from the command
          line'))
  parser.add_argument('gender')
  parser.add_argument('age', type=int)
  parser.add_argument('weight', type=int,
      help='in pounds')
  parser.add_argument('height', type=int,
      help='in inches')
  args = parser.parse_args()

  result = rmb(args.gender, args.age,
      args.weight, args.height)

  print('Your resting metabolic rate is
      {0}'.format(result))


def rmb(gender, age, weight, height):
  if gender == 'female':
    return (655 + 4.35 * weight + 4.7 *
        height - 4.7 * age) *  1.1
  else:
    return (66 + 6.25 * weight + 12.7 *
        height - 6.76 * age) * 1.1
```

```
  parser.add_argument('age', type=int)
  parser.add_argument('weight', type=int,
      help='in pounds')
  parser.add_argument('height', type=int,
      help='in inches')
  parser.add_argument('-k', '--kilos',
      action='store_true')
  args = parser.parse_args()

  result = rmb(
      args.gender, args.age, args.weight,
          args.height, use_kg=args.kilos)

  print('Your resting metabolic rate is
      {0}'.format(result))


def rmb(gender, age, weight, height,
    use_kg=False):
  w_const_female = 9.56 if use_kg else 4.35
  w_const_male = 13.75 if use_kg else 6.25
  if gender == 'female':
    return (655 + w_const_female * weight +
        4.7 * height - 4.7 * age) *  1.1
  else:
    return (66 + w_const_male * weight +
        12.7 * height - 6.76 * age) * 1.1
```

(a) Step 2.2



```
          line'))
  parser.add_argument('gender')
  parser.add_argument('age', type=int)
  parser.add_argument('weight', type=int,
      help='in pounds')
  parser.add_argument('height', type=int,
      help='in inches')
  parser.add_argument('-k', '--kilos',
      action='store_true')
  args = parser.parse_args()
```

```
  parser.add_argument('age', type=int)
  parser.add_argument('weight', type=int,
      help='in pounds')
  parser.add_argument('height', type=int,
      help='in inches')
  parser.add_argument(
      '-k', '--kilos', action='store_true',
      help='if true, then the weight given
          as input is interpreted as kg')
  args = parser.parse_args()
```

(b) Step 2.5

Figure 4: Modification steps for "Let users input weights in kilos" task (Task 2)

Task 3: Let users input height in meters

3.1. Let's get started with the height in meters feature.

1. Create a new branch `feat/meters` that diverges from `master` and switch to `feat/meters`
2. Open file `fit.py` and make this change

3.2. Alice just broke into your office! One of the main users of fit-cli reported a critical bug: no error is reported to the user if the user inputs a value that is not the string "female" or "male" for the gender. Since she wants you to fix this right away, we are going to fix this bug, and then go back to working on our features.

1. Switch back to `master`
2. Open file `fit.py` and make this change
3. Create a commit with the changes to `fit.py` and message "fix error checking bug"
4. Push the changes

Phew, the bug is now fixed. Now we are going to finish with our features.

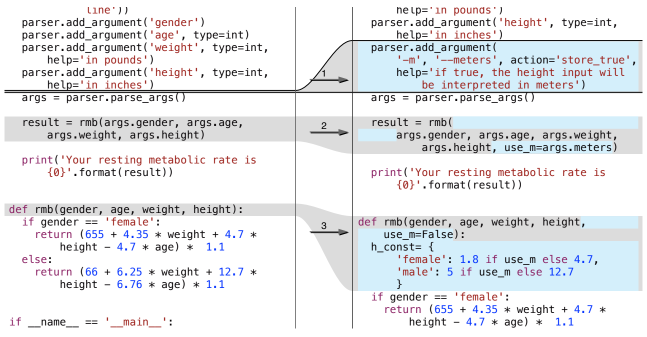
Table 8: "Let users input height in meters" task (Task 3)

the reason for this is that many thought that an execution of `gl branch -c feat/kilos` would not only create the branch but also switch to it.

A.1.3 Task 3: Let Users Input Height in Meters

Table 8 shows Task 3, Figure 5 shows the modification steps they had to perform as part of the task.

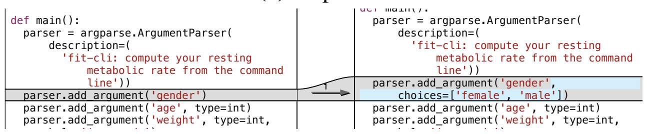
Seven participants completed the task successfully (72.72% success rate). Of the 3 that didn't, 2 created `feat/meters` diverging from `feat/kilos` instead of `master` (step 3.1.1) and never realized about it. The other failed to create the expected commit in step 3.2.3: the participant inadvertently carried over the uncommitted changes from the meters feature and included these in the bug fix on `master`.



```
          line'))
  parser.add_argument('gender')
  parser.add_argument('age', type=int)
  parser.add_argument('weight', type=int,
      help='in pounds')
  parser.add_argument('height', type=int,
      help='in inches')
  args = parser.parse_args()

  result = rmb(args.gender, args.age,
      args.weight, args.height)

  print('Your resting metabolic rate is
      {0}'.format(result))


def rmb(gender, age, weight, height):
  if gender == 'female':
    return (655 + 4.35 * weight + 4.7 *
        height - 4.7 * age) *  1.1
  else:
    return (66 + 6.25 * weight + 12.7 *
        height - 6.76 * age) * 1.1


if __name__ == '__main__':
```

```
      help='in pounds')
  parser.add_argument('height', type=int,
      help='in inches')
  parser.add_argument(
      '-m', '--meters', action='store_true',
      help='if true, the height input will
          be interpreted in meters')
  args = parser.parse_args()

  result = rmb(
      args.gender, args.age, args.weight,
          args.height, use_m=args.meters)

  print('Your resting metabolic rate is
      {0}'.format(result))


def rmb(gender, age, weight, height,
    use_m=False):
  h_const= {
      'female': 1.8 if use_m else 4.7,
      'male': 5 if use_m else 12.7
      }
  if gender == 'female':
    return (655 + 4.35 * weight + 4.7 *
        height - 4.7 * age) *  1.1
```

(a) Step 3.1.2



```
def main():
  parser = argparse.ArgumentParser(
      description=(
        'fit-cli: compute your resting
            metabolic rate from the command
            line'))
  parser.add_argument('gender')
  parser.add_argument('age', type=int)
  parser.add_argument('weight', type=int,
```

```
  parser = argparse.ArgumentParser(
      description=(
        'fit-cli: compute your resting
            metabolic rate from the command
            line'))
  parser.add_argument('gender',
      choices=['female', 'male'])
  parser.add_argument('age', type=int)
  parser.add_argument('weight', type=int,
```

(b) Step 3.2.2

Figure 5: Modification steps for "Let users input height in meters" task (Task 3)

For step 3.1.1, participants had to create a new branch `feat/meters` and switch to it. A key detail is that we asked them to make this branch diverge from `master`. Also note that after Task 2 participants were left with uncommitted changes and with the current branch being `feat/kilos`. There are two ways to complete this step. One is to switch to `master` and then create the new branch by doing `git branch feat/meters`. This would make the new branch point to the same commit `master` points to. Another alternative is to execute `git branch feat/meters master`, which would create the new branch diverging from `master` without doing the switch. All but one participant did the former and tried to do `git checkout master` but the execution failed. The problem is that the uncommitted changes in the working directory conflict with the changes in `master` and thus `checkout` fails. To work around this, 7 of the participants that completed the task resorted to creating a stash, the remaining one did an intermediate commit.

In step 3.2.1 participants had to switch to `master` to fix a bug. In this case, the uncommitted changes don't conflict with the changes in `master` so `checkout` would work. But the uncommitted changes are unrelated to the bug fix, so you wouldn't want them to follow you. To work around this, 6 participants created a stash, and 2 participants created an intermediate commit.

When using Gitless the success rate was higher (81.81%). The 2 participants that failed to complete the task created the `feat/meters` branch diverging from `feat/kilos` instead of `master`.

A.1.4 Task 4: Wrap With Features

Table 9 shows Task 4, Figure 6 shows the modification steps they had to perform as part of the task.

Task 4: Wrap with features

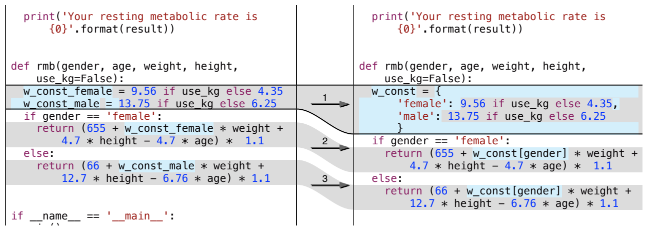
4.1. Bob wants us to wrap up the kilos feature so that it gets shipped with the next release. So instead of going back to working on the meters feature, we are going to leave that aside and finish with kilos.

1. Switch back to `feat/kilos` and go back to what we were doing before
2. Let's change to using a dictionary like Bob wanted. Open `fit.py` and make this change
3. Create a commit with the changes to `fit.py` and message "kilos improvements"
4. Update the kilos pull request: run the command `ut-pr-kilos-update`

4.2 While we wait to hear from Bob regarding the kilos feature, we are going to go back to finishing with the meters feature.

1. Switch to `feat/meters` and bring back changes
2. Now let's modify the computation of the rmb to use meters if the user specified so. Open `fit.py` and make this change.
3. Create a commit with the changes to `fit.py` and message "meters feature"
4. Bob will review our changes. Run the `ut-pr-meters-send` command to send a pull request.

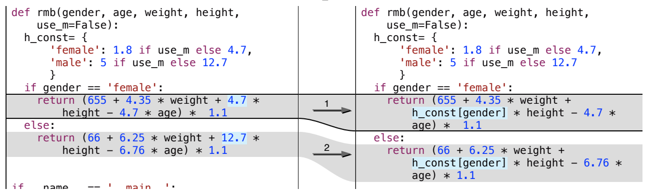
Table 9: "Wrap with features" task (Task 4)



```
  print('Your resting metabolic rate is
      {0}'.format(result))


def rmb(gender, age, weight, height,
    use_kg=False):
  w_const_female = 9.56 if use_kg else 4.35
  w_const_male = 13.75 if use_kg else 6.25
  if gender == 'female':
    return (655 + w_const_female * weight +
        4.7 * height - 4.7 * age) *  1.1
  else:
    return (66 + w_const_male * weight +
        12.7 * height - 6.76 * age) * 1.1


if __name__ == '__main__':
```

```
  print('Your resting metabolic rate is
      {0}'.format(result))


def rmb(gender, age, weight, height,
    use_kg=False):
  w_const = {
      'female': 9.56 if use_kg else 4.35,
      'male': 13.75 if use_kg else 6.25
      }
  if gender == 'female':
    return (655 + w_const[gender] * weight +
        4.7 * height - 4.7 * age) *  1.1
  else:
    return (66 + w_const[gender] * weight +
        12.7 * height - 6.76 * age) * 1.1
```

(a) Step 4.1.2



```
def rmb(gender, age, weight, height,
    use_m=False):
  h_const= {
      'female': 1.8 if use_m else 4.7,
      'male': 5 if use_m else 12.7
      }
  if gender == 'female':
    return (655 + 4.35 * weight + 4.7 *
        height - 4.7 * age) *  1.1
  else:
    return (66 + 6.25 * weight + 12.7 *
        height - 6.76 * age) * 1.1


if __name__ == '__main__':
```

```
def rmb(gender, age, weight, height,
    use_m=False):
  h_const= {
      'female': 1.8 if use_m else 4.7,
      'male': 5 if use_m else 12.7
      }
  if gender == 'female':
    return (655 + 4.35 * weight +
        h_const[gender] * height - 4.7 *
        age) *  1.1
  else:
    return (66 + 6.25 * weight +
        h_const[gender] * height - 6.76 *
        age) * 1.1
```

(b) Step 4.2.2

Figure 6: Modification steps for "Wrap with features" task (Task 4)

Six participants (out of 11) completed the task successfully (54.55% success rate). Of the 5 that failed, one skipped part 4.2, the others never applied the stashes that they saved in Task 3, so the commits they created for steps 4.1.3 and 4.2.3 were incomplete.

For bringing back changes in step 4.1.1, participants had to be careful of applying the correct stash, which was not the one in the top of the stack (thus `git stash pop` would apply the wrong one). Of all the participants that completed the task successfully one applied the wrong stash, but detected the mistake and ended up undoing the application of the stash by doing a `git reset fit.py` (so that `fit.py` is not marked

Task 5: Fixing conflicts

5.1. Bob accepted your kilos merge request, but now he's saying that the height pull request doesn't apply cleanly, and he's asking us to fix conflicts with `master` and update the pull request.

1. Update our local `master` branch with the new changes in the remote `master`
2. Switch back to `feat/meters` and rebase the changes from `master` (which will generate conflicts)
3. Let's start fixing conflicts. Open `fit.py` and make this change

5.2. Bob just broke into your office! Apparently we've been using the wrong shebang line all of this time. He wants you to fix this right away. We are going to fix this bug, and then go back to finish resolving the conflicts.

1. Switch to `master`
2. Open `fit.py` and make this change
3. Create a commit with the changes to `fit.py` and message "fix shebang"
4. Push

5.3. Let's finish fixing conflicts.

1. Switch back to `feat/meters` and go back to the state where we where before fixing the bug
2. Open `fit.py` and make this change
3. Finish with the rebase operation
4. Update the meters pull request: run the command `ut-pr-meters-update`

Table 10: "Fixing conflicts" task (Task 5)

as having conflicts anymore) followed by `git checkout fit.py` (to discard the changes in the working directory).

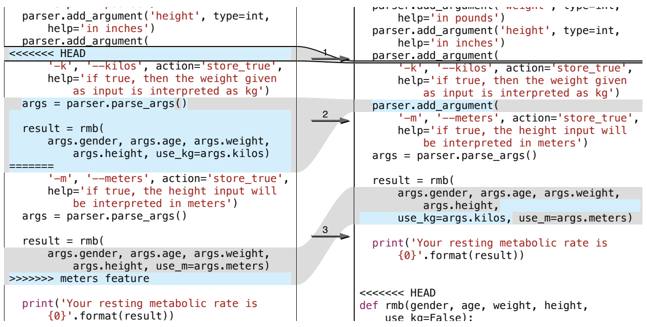
When using Gitless the success rate was higher (63.54%). Of the 4 participants that failed the task, one skipped part 4.1, another switched to `master` instead of to `feat/meters` in step 4.2.1, the other 2 created more commits than expected.

A.1.5 Task 5: Fixing Conflicts

Table 10 shows Task 5, Figure 7 shows the modification steps they had to perform as part of the task.

The challenge here was in step 5.2.1 where we asked the participant to switch to the `master` branch while in the middle of resolving conflicts. Many participants tried `git checkout master` that doesn't work because of the uncommitted changes, then tried `git stash` that doesn't work because of conflicts. Of the 4 participants that failed the task, 3 created an intermediate commit in the middle of the rebase to switch to `master` but when resuming the conflict resolution they never amend the commit, thus ending with a different history than the expected one. The other one that failed the task couldn't complete the rebase. (Task success rate was 54.55%.)

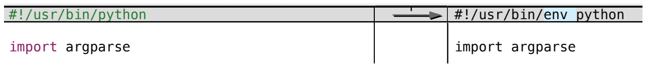
Of the 6 participants that completed the task successfully, 3 of them ended up redoing the changes they did in step 5.1.3. (Counting these as failures, the success rate would be as low as 27.27%). Of the other 3, one created a copy of the file out of the repository, aborted the rebase, restarted the rebase after the fix in `master`, and copied the file back again. The other two did an intermediate commit that they amended later.



```
  parser.add_argument('height', type=int,
      help='in inches')
  parser.add_argument(
<<<<<<< HEAD
      '-k', '--kilos', action='store_true',
      help='if true, then the weight given
          as input is interpreted as kg')
  args = parser.parse_args()

  result = rmb(
      args.gender, args.age, args.weight,
          args.height, use_kg=args.kilos)
=======
      '-m', '--meters', action='store_true',
      help='if true, the height input will
          be interpreted in meters')
  args = parser.parse_args()

  result = rmb(
      args.gender, args.age, args.weight,
          args.height, use_m=args.meters)
>>>>>>> meters feature

  print('Your resting metabolic rate is
      {0}'.format(result))
```

```
      help='in pounds')
  parser.add_argument('height', type=int,
      help='in inches')
  parser.add_argument(
      '-k', '--kilos', action='store_true',
      help='if true, then the weight given
          as input is interpreted as kg')
  parser.add_argument(
      '-m', '--meters', action='store_true',
      help='if true, the height input will
          be interpreted in meters')
  args = parser.parse_args()

  result = rmb(
      args.gender, args.age, args.weight,
          args.height,
      use_kg=args.kilos, use_m=args.meters)

  print('Your resting metabolic rate is
      {0}'.format(result))


<<<<<<< HEAD
def rmb(gender, age, weight, height,
    use_kg=False):
```

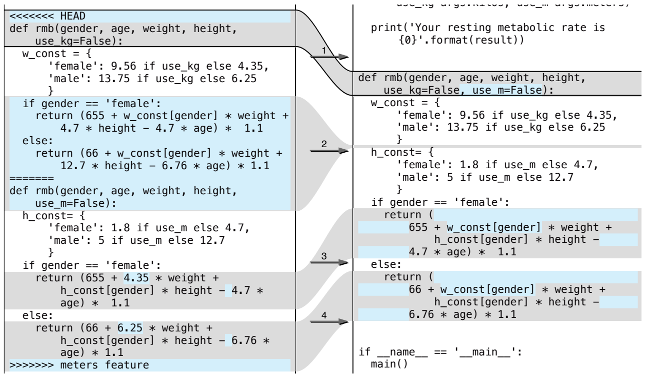
(a) Step 5.1.3



```
#!/usr/bin/python

import argparse
```

```
#!/usr/bin/env python

import argparse
```

(b) Step 5.2.2



```
<<<<<<< HEAD
def rmb(gender, age, weight, height,
    use_kg=False):
  w_const = {
      'female': 9.56 if use_kg else 4.35,
      'male': 13.75 if use_kg else 6.25
      }
  if gender == 'female':
    return (655 + w_const[gender] ∗ weight +
        4.7 ∗ height − 4.7 ∗ age) ∗  1.1
  else:
    return (66 + w_const[gender] ∗ weight +
        12.7 ∗ height − 6.76 ∗ age) ∗ 1.1
=======
def rmb(gender, age, weight, height,
    use_m=False):
  h_const= {
      'female': 1.8 if use_m else 4.7,
      'male': 5 if use_m else 12.7
      }
  if gender == 'female':
    return (655 + 4.35 ∗ weight +
        h_const[gender] ∗ height − 4.7 ∗
        age) ∗  1.1
  else:
    return (66 + 6.25 ∗ weight +
        h_const[gender] ∗ height − 6.76 ∗
        age) ∗ 1.1
>>>>>>> meters feature
```

```
  print('Your resting metabolic rate is
      {0}'.format(result))


def rmb(gender, age, weight, height,
    use_kg=False, use_m=False):
  w_const = {
      'female': 9.56 if use_kg else 4.35,
      'male': 13.75 if use_kg else 6.25
      }
  h_const= {
      'female': 1.8 if use_m else 4.7,
      'male': 5 if use_m else 12.7
      }
  if gender == 'female':
    return (
        655 + w_const[gender] ∗ weight +
            h_const[gender] ∗ height −
        4.7 ∗ age) ∗  1.1
  else:
    return (
        66 + w_const[gender] ∗ weight +
            h_const[gender] ∗ height −
        6.76 ∗ age) ∗ 1.1


if __name__ == '__main__':
  main()
```

(c) Step 5.3.2

Figure 7: Modification steps for "Fixing conflicts" task (Task 5)

With Gitless the success rate was 90.91%, the one participant that failed the task missed a few steps.

A.1.6 Task 6: Code Cleanup

Table 11 shows Task 6, Figure 8 shows the modification steps they had to perform as part of the task.

The success rate for this task was 63.64%. Two of the 4 participants that failed to complete the task ended the task in a detached head state with a commit that belongs to no branch at all. The other 2 missed one or more steps.

For step 6.2.1, one participant did a `git revert`, which creates a new commit that reverts the changes introduced by the last commit. This participant eventually realized that we wanted the last commit to disappear from the history and did a `checkout HEAD~2` to go back (which left the participant in detached head state). To prevent doing the last commit (step 6.2.3) in a detached head state the participant reattached the head and did a `git reset` instead. Another participant also did a `checkout` of the SHA and then reattached `HEAD` and did a `reset` instead. The remaining participants all did `reset` and never went into a detached head state.

The success rate was 81.82% when using Gitless. Of the 2 participants that failed the task, one skipped steps and ended the task in a very different state than the expected one. The other one forgot to `publish` changes. Since it is not possible to go in a detached head state in Gitless participants didn't have much trouble completing the task. Most of the time was spent figuring out how to change the head of the current branch (many tried looking for a flag in `checkout`).

Task 6: *Code cleanup*

6.1. It's time to clean up our code. This should be a fairly small task so we are going to keep it simple and work on `master`.

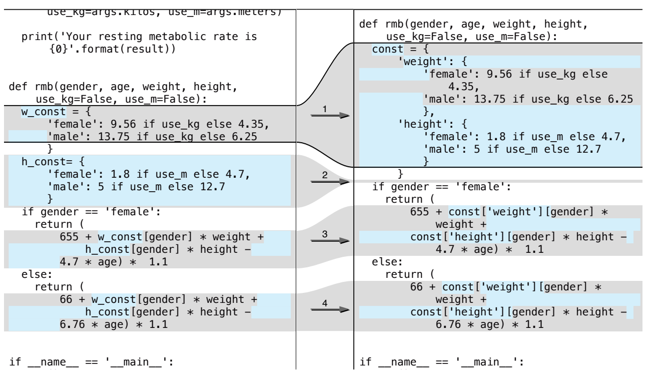
1. Switch to `master` and pull changes
2. Open `fit.py` and make this change
3. Create a commit with the changes to `fit.py` and message "join const dicts"

6.2. After a walk in the park you realized that the last commit might be wrong. So we are going to try something different.

1. Go back to the same state as before the commit
2. Open `fit.py` and make this change
3. Create a new commit with message "switch to using nt for consts"
4. Using `namedtuples` looks better than merging the dictionaries, so let's go ahead and push these changes

---

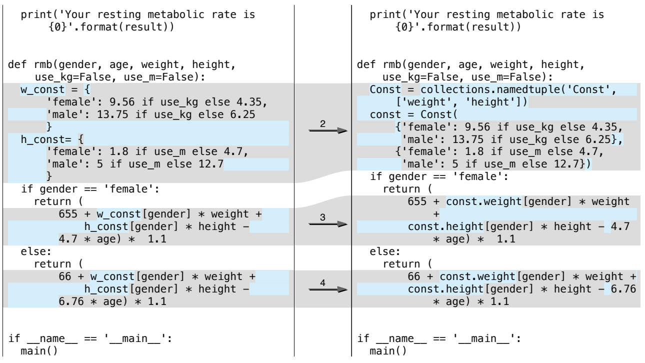
Table 11: "Code cleanup" task (Task 6)



```
      use_kg=args.kilos, use_m=args.meters)

  print('Your resting metabolic rate is
      {0}'.format(result))


def rmb(gender, age, weight, height,
    use_kg=False, use_m=False):
  w_const = {
      'female': 9.56 if use_kg else 4.35,
      'male': 13.75 if use_kg else 6.25
      }
  h_const= {
      'female': 1.8 if use_m else 4.7,
      'male': 5 if use_m else 12.7
      }
  if gender == 'female':
    return (
        655 + w_const[gender] ∗ weight +
            h_const[gender] ∗ height −
        4.7 ∗ age) ∗  1.1
  else:
    return (
        66 + w_const[gender] ∗ weight +
            h_const[gender] ∗ height −
        6.76 ∗ age) ∗ 1.1


if __name__ == '__main__':
```

```
def rmb(gender, age, weight, height,
    use_kg=False, use_m=False):
  const = {
      'weight': {
          'female': 9.56 if use_kg else
              4.35,
          'male': 13.75 if use_kg else 6.25
          },
      'height': {
          'female': 1.8 if use_m else 4.7,
          'male': 5 if use_m else 12.7
          }
      }
  if gender == 'female':
    return (
        655 + const['weight'][gender] ∗
            weight +
        const['height'][gender] ∗ height −
            4.7 ∗ age) ∗  1.1
  else:
    return (
        66 + const['weight'][gender] ∗
            weight +
        const['height'][gender] ∗ height −
            6.76 ∗ age) ∗ 1.1


if __name__ == '__main__':
```

(a) Step 6.1.2



```
  print('Your resting metabolic rate is
      {0}'.format(result))


def rmb(gender, age, weight, height,
    use_kg=False, use_m=False):
  w_const = {
      'female': 9.56 if use_kg else 4.35,
      'male': 13.75 if use_kg else 6.25
      }
  h_const= {
      'female': 1.8 if use_m else 4.7,
      'male': 5 if use_m else 12.7
      }
  if gender == 'female':
    return (
        655 + w_const[gender] ∗ weight +
            h_const[gender] ∗ height −
        4.7 ∗ age) ∗  1.1
  else:
    return (
        66 + w_const[gender] ∗ weight +
            h_const[gender] ∗ height −
        6.76 ∗ age) ∗ 1.1


if __name__ == '__main__':
  main()
```

```
  print('Your resting metabolic rate is
      {0}'.format(result))


def rmb(gender, age, weight, height,
    use_kg=False, use_m=False):
  Const = collections.namedtuple('Const',
      ['weight', 'height'])
  const = Const(
      {'female': 9.56 if use_kg else 4.35,
       'male': 13.75 if use_kg else 6.25},
      {'female': 1.8 if use_m else 4.7,
       'male': 5 if use_m else 12.7})
  if gender == 'female':
    return (
        655 + const.weight[gender] ∗ weight
            +
        const.height[gender] ∗ height − 4.7
            ∗ age) ∗  1.1
  else:
    return (
        66 + const.weight[gender] ∗ weight +
        const.height[gender] ∗ height − 6.76
            ∗ age) ∗ 1.1


if __name__ == '__main__':
  main()
```

(b) Step 6.2.2

Figure 8: Modification steps for "Code cleanup" task (Task 6)

References

[1] B. Appleton, S. P. Berczuk, R. Cabrera, and R. Orenstein. Streamed lines: Branching patterns for parallel software development. 1998.

[2] E. T. Barr, C. Bird, P. C. Rigby, A. Hindle, D. M. German, and P. Devanbu. Cohesive and isolated development with branches. In *Proceedings of the International Conference on Fundamental Approaches to Software Engineering*. Springer, April 2012. URL `http://research.microsoft.com/apps/pubs/default.aspx?id=157290`.

[3] S. Bennett. 10 things I hate about git, 2012. URL `http://stevebennett.me/2012/02/24/10-things-i-hate-about-git/`.

[4] M. Biazzini, M. Monperrus, and B. Baudry. On analyzing the topology of commit histories in decentralized version control systems. In *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on*, pages 261–270, Sept 2014. doi: 10.1109/ICSME.2014.48.

[5] C. Bird and T. Zimmermann. Assessing the value of branches with what-if analysis. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 45:1–45:11, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1614-9. doi: 10.1145/2393596.2393648. URL `http://doi.acm.org/10.1145/2393596.2393648`.

[6] C. Brindescu, M. Codoban, S. Shmarkatiuk, and D. Dig. How do centralized and distributed version control systems impact software changes? In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 322–333, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2756-5. doi: 10.1145/2568225.2568322. URL `http://doi.acm.org/10.1145/2568225.2568322`.

[7] F. P. Brooks. *The Mythical Man-Month: Essays on Software Engineering*. Addison-Wesley Professional, 1995. ISBN 978-0-201-83595-3.

[8] F. P. Brooks. *The Design of Design: Essays from a Computer Scientist*. Addison-Wesley Professional, 2010. ISBN 978-0-201-36298-5.

[9] S. Chacon and B. Straub. *Pro Git*. Apress, 2 edition, 2014. ISBN 978-1484200773. URL `http://git-scm.com/book`.

[10] L. Church, E. Söderberg, and E. Elango. A case of computational thinking: The subtle effect of hidden dependencies on the user experience of version control. 2014. PPIG 2014 - 25th Annual Workshop.

[11] B. de Alwis and J. Sillito. Why are software projects moving from centralized to decentralized version control systems? In *Proceedings of the 2009 ICSE Workshop on Cooperative and Human Aspects on Software Engineering*, CHASE '09, pages 36–39, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-1-4244-3712-2. doi: 10.1109/CHASE.2009.5071408. URL `http://dx.doi.org/10.1109/CHASE.2009.5071408`.

[12] S. Garfinkel, D. Weise, and S. Strassmann. *The UNIX Hater's Handbook: The Best of UNIX-Haters On-line Mailing Reveals Why UNIX Must Die!* IDG Books Worldwide, Inc., June 1994. ISBN 978-1-56884-203-5. URL `http://web.mit.edu/~simsong/www/ugh.pdf`.

[13] G. Gousios, M. Pinzger, and A. v. Deursen. An exploratory study of the pull-based software development model. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 345–355, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2756-5. doi: 10.1145/2568225.2568260. URL `http://doi.acm.org/10.1145/2568225.2568260`.

[14] T. R. G. Green and M. Petre. Usability analysis of visual programming environments: a "cognitive dimensions" framework. *Journal of Visual Languages & Computing*, 7(2): 131–174, 1996.

[15] A. Henderson and J. Johnson. *Conceptual Models: Core to Good Design*. Synthesis Lectures on Human-Centered Informatics. Morgan & Claypool Publishers, 2011. ISBN 978-1608457496.

[16] D. Jackson. Towards a theory of conceptual design for software. In *2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)*, Onward! 2015, pages 282–296, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3688-8. doi: 10.1145/2814228.2814248. URL `http://doi.acm.org/10.1145/2814228.2814248`.

[17] Y. Jiang, B. Adams, and D. M. German. Will my patch make it? and how fast?: Case study on the linux kernel. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13, pages 101–110, Piscataway, NJ, USA, 2013. IEEE Press. ISBN 978-1-4673-2936-1. URL `http://dl.acm.org/citation.cfm?id=2487085.2487111`.

[18] J. Loeliger and M. McCullough. *Version Control with Git: Powerful Tools and Techniques for Collaborative Software Development*. O'Reilly Media, second edition, 2012. ISBN 978-1-4493-1638-9.

[19] K. Muşlu, C. Bird, N. Nagappan, and J. Czerwonka. Transition from centralized to decentralized version control systems: A case study on reasons, barriers, and outcomes. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 334–344, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2756-5. doi: 10.1145/2568225.2568284. URL `http://doi.acm.org/10.1145/2568225.2568284`.

[20] J. Nielsen and R. Molich. Heuristic evaluation of user interfaces. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '90, pages 249–256, New York, NY, USA, 1990. ACM. ISBN 0-201-50932-6. doi: 10.1145/97243.97281. URL `http://doi.acm.org/10.1145/97243.97281`.

[21] B. O'Sullivan. *Mercurial: The Definitive Guide*. O'Reilly Media, 2009. ISBN 978-0596800673.

[22] S. Perez De Rosso and D. Jackson. What's wrong with git? a conceptual design analysis. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! 2013, pages 37–52, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2472-4. doi: 10.1145/2509578.2509584. URL `http://doi.acm.org/10.1145/2509578.2509584`.

[23] S. Phillips, J. Sillito, and R. Walker. Branching and merging: An investigation into current version control practices. In

Proceedings of the 4th International Workshop on Cooperative and Human Aspects of Software Engineering, CHASE '11, pages 9–15, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0576-1. doi: 10.1145/1984642.1984645. URL http://doi.acm.org/10.1145/1984642.1984645.

[24] C. M. Pilato, B. Collins-Sussman, and B. W. Fitzpatrick. *Version Control with Subversion*. O'Reilly Media, 2008. ISBN 0596510330. URL http://svnbook.red-bean.com/.

[25] B. Pollack. Unorthodocs: Abandon your DVCS and return to sanity, 2015. URL http://bitquabit.com/post/unorthodocs-abandon-your-dvcs-and-return-to-sanity/.

[26] P. C. Rigby, E. T. Barr, C. Bird, P. Devanbu, and D. M. German. What effect does distributed version control have on oss project organization? In *Proceedings of the 1st International Workshop on Release Engineering*, RELENG '13, pages 29–32, Piscataway, NJ, USA, 2013. IEEE Press. ISBN 978-1-4673-6441-6. URL http://dl.acm.org/citation.cfm?id=2663360.2663368.

[27] C. Rodríguez-Bustos and J. Aponte. How distributed version control systems impact open source software projects. In *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories*, MSR '12, pages 36–39, Piscataway, NJ, USA, 2012. IEEE Press. ISBN 978-1-4673-1761-0. URL http://dl.acm.org/citation.cfm?id=2664446.2664452.

[28] C. Walrad and D. Strom. The importance of branching models in scm. *Computer*, 35(9):31–38, Sep 2002. ISSN 0018-9162. doi: 10.1109/MC.2002.1033025.