A Conceptual Design Analysis of Git

by

Santiago Perez De Rosso

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of
Master of Science
at the
MASSACHUSETTS INSTITUTE OF TECHNOLOGY

Feb 2015

© Massachusetts Institute of Technology 2015. All rights reserved.

Signature redacted

Author ..................................................
Department of Electrical Engineering and Computer Science
January 30, 2015

Signature redacted

Certified by...
Daniel Jackson
Professor
Thesis Supervisor

Signature redacted

Accepted by .......
Professor Leslie A. Kolodziejski
Chair, Department Committee on Graduate Students
A Conceptual Design Analysis of Git

by

Santiago Perez De Rosso

Submitted to the Department of Electrical Engineering and Computer Science on January 30, 2015, in partial fulfillment of the requirements for the degree of Master of Science

Abstract

It is commonly asserted that the success of a software development project, and the usability of the final product, depend on the quality of the concepts that underlie its design. Yet this hypothesis has not been systematically explored by researchers, and conceptual design has not played the central role in the research and teaching of software engineering that one might expect.

As part of a new research project to explore conceptual design, we are engaging in a series of case studies. This thesis reports on our case study on Git, a popular—yet sometimes puzzling—version control system. In an attempt to understand the root causes of its complexity, we analyze its conceptual model and identify some undesirable properties; we then present a reworking of the conceptual model that forms the basis of Gitless, our redesign of Git.

Thesis Supervisor: Daniel Jackson
Title: Professor
Acknowledgments

This research was part of a collaboration between MIT and SUTD (the Singapore University of Technology and Design), and was funded by a grant from SUTD’s International Design Center.

Thank you to Daniel Jackson, whose expert guidance made this work possible.

Thanks also to Marcelo Frias, who introduced me to the world of research.

To my friends, colleagues, and family.

And to Michelle, for her love and support.
Contents

1 Introduction ............................................................................. 15
  1.1 Background and Motivation ................................................. 15
  1.2 Choice of Git as Case Study ................................................. 17

2 Elements of Conceptual Design .............................................. 19
  2.1 Criteria for Conceptual Design ............................................ 20

3 Purposes for Version Control ................................................ 23
  3.1 Data Management ............................................................ 23
  3.2 Change Management ......................................................... 24
  3.3 Collaboration .................................................................... 25
  3.4 Parallel Development ......................................................... 25
  3.5 Disconnected Development ............................................... 26

4 An Overview of Git ............................................................... 27
  4.1 Distributed ................................................................. 27
  4.2 Recording Changes .......................................................... 29
  4.3 Branching ...................................................................... 30
  4.4 Tagging ....................................................................... 30
  4.5 Integrating Changes ......................................................... 31
  4.6 Syncing with other Repositories ........................................ 34
  4.7 History ....................................................................... 37
### 5 A Conceptual Model of Git

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>The Abstract State Space</td>
<td>39</td>
</tr>
<tr>
<td>5.1.1</td>
<td>Notation</td>
<td>39</td>
</tr>
<tr>
<td>5.1.2</td>
<td>Discussion</td>
<td>42</td>
</tr>
<tr>
<td>5.1.3</td>
<td>Other Constraints</td>
<td>44</td>
</tr>
<tr>
<td>5.2</td>
<td>Explanations</td>
<td>44</td>
</tr>
<tr>
<td>5.2.1</td>
<td>Tracked and Untracked</td>
<td>44</td>
</tr>
<tr>
<td>5.3</td>
<td>Purpose Graph</td>
<td>48</td>
</tr>
<tr>
<td>5.3.1</td>
<td>Stash</td>
<td>48</td>
</tr>
</tbody>
</table>

### 6 Analysis of Git’s Conceptual Model

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1</td>
<td>Commit</td>
<td>51</td>
</tr>
<tr>
<td>6.2</td>
<td>File</td>
<td>52</td>
</tr>
<tr>
<td>6.3</td>
<td>File Path Classifications</td>
<td>52</td>
</tr>
<tr>
<td>6.4</td>
<td>Staging Area</td>
<td>53</td>
</tr>
<tr>
<td>6.5</td>
<td>Branch</td>
<td>54</td>
</tr>
<tr>
<td>6.6</td>
<td>Stash</td>
<td>55</td>
</tr>
</tbody>
</table>

### 7 Gitless

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.1</td>
<td>Overview</td>
<td>58</td>
</tr>
<tr>
<td>7.2</td>
<td>Conceptual Model</td>
<td>59</td>
</tr>
<tr>
<td>7.3</td>
<td>Gitless versus Git</td>
<td>59</td>
</tr>
<tr>
<td>7.3.1</td>
<td>A Compelling Concept of Branching</td>
<td>64</td>
</tr>
<tr>
<td>7.3.2</td>
<td>Controllable File Path Classifications</td>
<td>65</td>
</tr>
<tr>
<td>7.3.3</td>
<td>No Interfering Staging Area</td>
<td>65</td>
</tr>
<tr>
<td>7.4</td>
<td>Gitless versus Mercurial</td>
<td>66</td>
</tr>
</tbody>
</table>

### 8 Related Work

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.1</td>
<td>Usability and Conceptual Models</td>
<td>69</td>
</tr>
<tr>
<td>8.2</td>
<td>Conceptual Integrity</td>
<td>70</td>
</tr>
<tr>
<td>8.3</td>
<td>Conceptual Modeling</td>
<td>71</td>
</tr>
</tbody>
</table>
9 Future Work
List of Figures

2-1 Uncompelling and compelling faucets ........................................... 20
2-2 Controllable concepts in Keynote .................................................. 21
4-1 Workflows commonly used in Git .................................................... 28
4-2 Merge ......................................................................................... 32
4-3 Rebase ......................................................................................... 32
4-4 Cherry-pick .................................................................................. 33
4-5 Syncing with another repository .................................................... 35
5-1 The abstract state space of Git’s conceptual model ......................... 41
5-2 Git’s purpose graph ....................................................................... 49
7-1 The abstract state space of Gitless’s conceptual model ..................... 60
7-2 Gitless’s purpose graph ................................................................... 63
List of Tables

5.1 Definitions for Git’s concepts ........................................... 45
5.2 Motivating purposes for Git’s concepts .............................. 46

7.1 Definitions for Gitless’s concepts ................................. 61
7.2 Motivating purposes for Gitless’s concepts .................. 62
Chapter 1

Introduction

1.1 Background and Motivation

What are the principles that should guide the design of software applications? Amongst the many that have been proposed, we can distinguish two classes: usability principles, which aim to make software easier to use, and software engineering principles, which aim to make the software easier to build and maintain. Perhaps surprisingly, the same software engineering principles determine not only the development process but also its outcome, in the performance of the application (how fast it runs and how demanding it is in terms of resources of memory, bandwidth and so) and its reliability (how likely it is to fail).

The two classes of principles have rarely come into contact. Usability principles tend to focus on the user interface: in particular, on its appearance and structure. Take a look, for example, at the lists of heuristics published by the Nielsen Norman Group [27], which include “visibility of system status,” “speaking the user’s language,” and “following platform conventions.” In contrast, while usability experts have focused on externals, software engineers have usually restricted themselves to internals. Thus software engineering principles talk about modularization or encapsulation in the code [31], about coupling between code modules [32], and about combining power and simplicity in programmer interfaces [23].

The premise of the research program this case study is part of [20] is that this
The essence of software design, in our view, resides in the conceptual structure embodied in the code and expressed through the user interface. This conceptual structure is neither about code nor about user interface elements; rather, it is about the basic concepts that both user and developer have in mind, whether they are interacting with the system or building it. While the user interface and even the entire codebase of an application may change from version to version, it is the concepts that remain largely the same. In Microsoft Word, for example, the key concepts include that of “paragraph” and “style”; indeed, most users are not even aware of the fact that both of these represented a major innovation when first introduced (having been brought by Charles Simonyi from the Bravo development at Xerox PARC). The concept of a paragraph is so simple that it needs little explanation (although it is hardly uncontroversial; the difficulty of integrating sections into Word is a direct result of the paragraph-based structure, in contrast to the hierarchical structure of TeX, for example). More subtle and complex concepts are sometimes needed; witness the enabling concept of “relative references” in spreadsheets, or “layers” and “masks” in Adobe Photoshop.

The primacy of concepts is arguably not a new idea at all. In his influential book on software engineering management, *The Mythical Man Month* [4], Fred Brooks described conceptual integrity as “the most important consideration in system design,” and in a 20-year anniversary reissue of the book [5] stood by this claim, stating: “I am more convinced than ever. Conceptual integrity is central to product quality.” Usability experts too (such as Donald Norman) have recognized that usability requires a match between the conceptual model in the minds of the user and the designer [28]. Furthermore, there is a long tradition of industrial designers recognizing that the physical surface of an artifact should not be the focus of its design, but should instead emerge from a deeper understanding, with the design proceeding “from the inside out” [10].

What is new in our research project is our explicit focus on the conceptual structure as a central object of intentional design. Authors such as Brooks and Norman have recognized the centrality of concepts, but have seen the conceptual structure of
a design more as the accidental result of the design of concrete parts than as a design focus in its own right.

The touchstone of our research is how effective our approach will be in practice in improving the design of software applications. A major thrust of our project is thus a series of case studies in which we take existing applications, model, analyze and redesign them in an attempt to show that bad concepts explain surprising behaviour, and that a better conceptual structure results in greater usability. This thesis presents our case study on Git, a popular—yet sometimes puzzling—version control system. An initial report of this case study appears in [33].

1.2 Choice of Git as Case Study

We picked Git for several reasons. First, it is a popular and widely used program, so the case study would likely be of interest to a larger audience than a study of a more obscure application. Second, Git is powerful and feature rich; this makes redesign more challenging, and reduces the risk that the case study is a straw-man with little relevance to well designed applications. Third, being so powerful, a conceptual redesign of Git could be easily implemented using the existing code base, with a wrapper that acts as a kind of conceptual “impedance matcher.” This enables us to rapidly prototype tools that represent different points in the concept design space and try them out with users (to misquote von Moltke “no design survives first contact with the user”). Last but not least, it is our impression that Git is far more complicated than it needs to be, and that many of the difficulties that users face in using Git can be attributed to flaws in its conceptual model.
Chapter 2

Elements of Conceptual Design

This chapter covers the aspects of our theory for conceptual design that we apply in this case study. While there is some overlap in our vocabulary with that of other authors who have (to varying degrees) covered this topic before, what we understand by, for example, “conceptual design,” or “conceptual model” might not necessarily agree with these other views. We thus recommend the reader to read this chapter with a clean slate of mind. (More about the overlap with other works in Chapter 8.)

“Conceptual design,” to us, is the creation of a suitable conceptual model, intended to serve a particular purpose.

A “conceptual model” is a state machine, where concepts correspond to state components, and actions produce transitions between abstract states. Concepts are the essential constructs and notions that the system deals with. Concepts have psychological content; they correspond to how users think about the application. To connect the abstract state components (which by themselves would lack any psychological content) with the user’s understanding of them as concepts we use the notion of (what we are calling) an “explanation”: a definition (i.e., what is the concept?) accompanied by its motivating purpose (i.e., why does the concept exist?).

A “purpose” is a desired result, about the product, which is concrete and measurable. It is not a way to achieve a result, nor a code or design detail. Knowing

---

the intended purpose of a system is crucial for the correct analysis of its conceptual model.

### 2.1 Criteria for Conceptual Design

**Compelling.** We say a concept is “compelling” if it has a simple operational principle and is motivated by one purpose. A physical example of a violation of this criterion is the faucet of Fig. 2-1a. The faucet of Fig. 2-1a has a rather complex operation principle (which constitutes a violation to the compelling criterion). Turning any of the handles results in both a change in temperature and flow, which means that a subsequent adjustment is needed (turning the other handle) to maintain the other constant. In contrast, in the faucet of Fig. 2-1b, the temperature and flow can be adjusted independently of the other, by turning the handle up or down (to adjust the flow), or [anti]clockwise (to adjust the temperature). This faucet has a simple operation principle (to adjust flow, turn handle up or down; to adjust temperature, rotate handle), and is motivated by two subpurposes (temperature and flow) with one control for each; it is therefore a compelling faucet.

**Controllable.** We say a concept is “controllable” if it fulfills its purpose without interference from other concepts. An example of a violation to this criterion is in Gmail (Google’s email client). In Gmail, you can assign a label to an email, but multiple emails can be grouped together into a conversation. Then applying a label to an individual email is no longer possible, the label will apply to the conversation instead (and thus to every email in that conversation). It is therefore not possible to
have two different emails with different labels that belong to the same conversation. This shows the interference of conversations with labels.\(^2\) Keynote (Apple’s presentation software) has a good example of controllable concepts (skipping and hierarchy). In Keynote, as shown in Fig. 2-2a, you can skip slides (a skipped slide will not appear in the slideshow). It also lets you, as shown in Fig. 2-2b, create a hierarchy for your slides (i.e., a slide can have one or more parents). What makes these two concepts controllable is the fact they do not interfere with each other: skipping a slide will skip that slide only, without interfering with the hierarchical structure and, conversely, changing the hierarchical structure of a presentation does not affect which slides are skipped. Fig. 2-2c shows this non-interference in action.

**Conventional.** We say a concept is “conventional” if it reuses existing concepts when applicable. These pre-existing concepts could come from other similar products, or from within the same product. Take the concept of the trashcan (which allows undoing deletions); it is the same in most operating systems. This is an example of a conventional concept.

\(^2\)This example was contributed by Eunsuk Kang.
Chapter 3

Purposes for Version Control

We develop a small set of purposes which (we believe) capture the essence of modern version control. These are based on our own experience and on popular references on the web that refer to common VCS’s use cases. Our list is high level, and not concerned with specific aspects of Git. We classify the purposes into the following categories: data management (Sec. 3.1), change management (Sec. 3.2), collaboration (Sec. 3.3), parallel development (Sec. 3.4), and disconnected development (Sec. 3.5).

3.1 Data Management

Purpose 1 Make a set of changes to files persistent
This refers to the ability of saving changes, so that they can be later retrieved in case of failure.

The category of data management corresponds to the idea of a “backup,” its purpose being to be able to recover data if it is lost (for example, by corruption or accidental deletion).

Most backups systems tend to also have a way of recovering old versions of the data you save (and not just the last one). At first, it might seem like this is no longer about fault tolerance, but instead about keeping a history. But being able to recover old versions of the data you save is also about tolerating failure, albeit one
of a different kind: where all (or most of) the edits to a document done after some point in time are “wrong” (and it is thus convenient to revert back to an old version of the document).

### 3.2 Change Management

**Purpose 2 Represent/record coherent points**

This refers to the ability of somehow recording that, at certain point in time, the state of the repository is “coherent.” What constitutes a coherent repository usually depends on each team (or maybe even on each contributor). It might mean that all tests pass, the code compiles, or that it provides a stable point from where to base work on (it will not disappear or change). Usual coherent points in the history of a repository include those in which a release was made, a feature was finalized, or a bug was fixed.

**Purpose 3 Group logically related changes together**

This refers to the ability of grouping a set of changes made to a set of files together into one logical group. If it were not for this, making sense out of the collection of changes made to all files would be very difficult.

Different to data management is *change management*. VCSs tend to entangle these together; but change management is what distinguishes them from backup systems. The assumption is that a VCS will give you not only the means to save and restore versions of files (data management), but also the means to manage (administer, organize) *change*.

(The main consequence of this entanglement is the notion of commit, which as a result of doing too many things, it is a subpar solution for each—but more on this later in the analysis chapter.)

The two purposes we identify in this category, “represent/record coherent points” and “group logically related changes together,” correspond to saying (in another
words), “at this point in time, the repository is in a coherent state,” and “all of these changes belong together” respectively.

3.3 Collaboration

**Purpose 4** *Sync (synchronize) changes of collaborators*

Any significant project will usually have multiple collaborators working on it simultaneously. Being able to easily sync (some or all of) these changes is one of the purposes of a VCS. This involves downloading changes from other collaborators, and making local changes available so that others can download them. It also involves identifying conflicts, and giving a way for the user to manually fix them.

3.4 Parallel Development

**Purpose 5** *Switch between parallel lines*

Underlying the idea of parallel development is that one can actually do work on each of the active lines in parallel. This purpose refers to the ability of switching between lines to do work.

**Purpose 6** *Sync changes of parallel lines*

Like with collaboration, the changes of parallel lines also usually need to be synchronized.

In some cases, parallel development is about maintaining multiple parallel releases of the same software (where each active release has its own line): while new features might go only to the line corresponding to the current release, bug fixes and security patches would also be applied to the other active releases. In other cases, parallel lines of development are maintained as an organizational mechanism to keep the development of major features isolated from one another.
3.5 Disconnected Development

**Purpose 7** *Do work in disconnected mode*

This refers to the ability of working without requiring an internet connection and without requiring coordination with other collaborators.

There is more to disconnected development than being able to work without an internet connection. It also refers to the ability of disconnecting yourself from your collaborators: being able to work while ignoring changes made by others until necessary. A crucial difference between CVCSs (Centralized Version Control Systems) and DVCSs (Distributed Version Control Systems) is how well they fulfill this purpose. In CVCSs, 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 DVCSs you can delay connecting (to the rest of your collaborators or to the internet) until you want to share your changes with others. (This distinction will be made more clear in the next chapter where we give an overview of Git.)
Chapter 4

An Overview of Git

Before diving into our model of Git, we begin by giving an informal introduction to the Git VCS. This should help the reader become familiar with the key functionality and concepts behind Git.

4.1 Distributed

Unlike in CVCSs (such as Subversion\(^1\)), in Git (like in other DVCSs), each user has her own 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—if the user has permissions\(^2\) to do so).

It is common to designate one of the repositories as a shared “remote” repository that acts as a hub for synchronizations between the “local” repositories of individual users. In this way, the benefits of both centralized and local repositories are obtained. But the distributed nature of Git (where each user has a local copy of the repository) enables more complex workflows, usually best suited (and adopted) for

\(^1\)https://subversion.apache.org.

\(^2\)Git *per se* does not manage permissions; this is instead handled out-of-band, by limiting access to the server hosting the Git repository. For example, by giving ssh access to only some users, or if the repository is hosted on GitHub, GitHub has a permissions layer built-in that allows access control for other GitHub users to be set up.
the development of open source projects. In the most common of these, instead of directly pushing to a central repository, individual developers send “pull requests” to the owner(s) of some blessed repository who (optionally) pull the changes in. Developers keep their repositories updated by pulling changes from this blessed repository. There is nothing special about a blessed repository other than the fact that it was designated to be the canonical reference for the project. Fig. 4-1 illustrates these two workflows, usually referred to as “centralized,” and “integration-manager.”\(^3\) The arrows denote the direction of the flow of data and the circle is where the action originates.

So-called “bare” repositories, which only have the version control information and no working directory, are usually used for the “hub” repositories, since no user is directly working on them (but instead working on their own local repositories and only using the hub for synchronization). The purpose of the distinction between repositories that have a working directory (non-bare repositories) and those that do

\(^3\)A third popular variant, “dictator-luitenants,” works similar to “integration-manager” but with more layers of integrators. See [http://git-scm.com/about/distributed](http://git-scm.com/about/distributed) for more on this.
not (bare repositories) seems to be purely motivated to avoid confusing behavior: if
the user is working on a repository and some other user pushes to it, the underlying
base which the user is working on could change without notice. Therefore, push
complains if the user tries to push changes to a non-bare repository.

4.2 Recording Changes

Between the user's working directory and the local repository, Git interposes a "staging
area," also called the "index." All commits are made via this intermediate area. Thus
the standard workflow is first to make copies of files to the staging area (with the add
command), and then to commit them to the repository (with the commit command).
Explanations of Git use the term "tracked files" to refer to files that have been staged.
This confuses novices, since such files are tracked only in the sense that the status
command will notice that changes to them have not been committed. Contrary to
initial expectation, 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 the add command was called), and not the latest one.

Worse, tracked files may not have a corresponding version in the staging area.
Following a commit, a file that had been previously staged remains tracked, but the
version in the staging area is removed. The term "staged file" often used interchange-
ably with "tracked file" is thus subtly different: in this case, we have a file that is
tracked but no longer staged.

Finally, 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
the committing of non-source files. But tracked files cannot be ignored; to ignore a
tracked file one has mark it as "assume unchanged." This "assume unchanged" file will
not even be recognized by the add command; to make it tracked again this marking
has to be removed.
4.3 Branching

A branch is a named pointer to a commit. Git provides the illusion that at any one time the user is working on some branch of development by always updating the branch pointer on commit to point to the newly created commit (thus a branch is also usually defined as a movable, named pointer to a commit). But as many novices eventually find out (to their surprise), it is possible to be doing work that belongs to no branch at all. Some operations might send the user down a rabbit-hole known as a “detached HEAD” state. This happens because 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.

Yet branches are a very popular resource for keeping development tasks separate; and repositories usually have several of them. Switching to another branch enables the user to put aside one development task and work on another (for example, to pursue the implementation of a particular feature, or fix a particular bug). But switching branches is a complex matter, because, although the branches are maintained separately in the repository, there is only one working directory and one staging area. As a result, when switching branches, files may be unexpectedly overwritten. Git fails with an error if there are any conflicting changes, effectively preventing the user from switching branches in these cases. To mitigate this problem, Git provides a way to save versions of files to yet another storage area, called the “stash,” using a special command issued prior to switching branches.

4.4 Tagging

A tag is a named pointer to a commit. But unlike a branch, a tag pointer does not get updated on commit (it is static). Tags are usually used to note that certain commit is special in some way. For example, a new release might have been created at the commit pointed to by a tag (names like “v1.0” and so are common for tags).
4.5 Integrating Changes

Git provides three ways of integrating changes: merge, rebase and cherry-pick. Their purpose is to unify changes from two (or more) branches. It does not matter whether the branches are local or remote; all changes are brought together through one of these three basic operations. For example, pull is no more than a shorthand for fetch (download changes from a remote) followed by merge (or rebase if the --rebase flag is used). And push can be in turn seen as another fetch followed by a “fast-forward merge” (discussed below) from the perspective of the target repository (the one where changes are being pushed to).

Merge. The merge of two branches (usually) results in a new (merge) commit with two parents. But an execution of the merge command might not actually create a merge commit. It could do a “fast-forward merge” instead, where the current branch is updated to point to the commit of the other branch. This happens when the other branch is ahead of the current branch—the commit pointed to by the current branch is an ancestor of the commit pointed to by the other branch. (It is possible to override this “fast-forward” behavior with the --no-ff flag of merge.) Fig. 4-2 illustrates a regular merge and a “fast-forward” one. In both cases, the current branch (i.e., the one pointed to by HEAD), master, is merged with the other branch develop.

It is worth noting that it is possible to merge more than two branches together in one merge operation. In this case, the resulting merge commit would have more than two parents.

Rebase. Rebase changes the base of the current branch to the other branch. What constitutes the “base” of the current branch depends on the other branch: it is the set of commits both branches have in common (i.e., from their least common ancestor up to the root). So rebase looks for the least common ancestor (the divergent point between both branches), and it applies (replays) all the commits from the current branch that follow the common ancestor after the other branch commits. Fig. 4-3 illustrates a rebasing of the current branch master to a branch develop. What used
Figure 4-2: Merge

(a) Regular merge

(b) "Fast-forward" merge

Figure 4-3: Rebase

to be commit B becomes B'. This is because, even if no conflicts were fixed by the user, B' is still a different commit from B. (A commit in Git is a complete snapshot of the repository at that point in time, and B' is a different snapshot than B since it additionally has the changes introduced by C and D.)

The rebase command also has an interactive feature that allows the user to customize how the commits should be applied. For example, it is possible to squash multiple commits into one, split a commit into multiple commits and so. Users thus sometimes use rebase as a tool to "cleanup history"; this consists of rebasing the current branch with its remote counterpart. Since the remote branch points to an ancestor of the commit pointed to by the current branch the rebase is a trivial
one (in the sense that there cannot be conflicts—it is not really changing the base), and the interactive features of the rebase command allows the user to format the yet-unpushed changes.

**Cherry-pick.** Cherry-pick allows the user to specify a set of commits from another branch to be applied onto the current branch. Fig. 4-4 illustrates a cherry-pick of commit C from branch develop onto branch master. Note the likeness of cherry-pick to rebase: if the user was to cherry-pick both C and D, the result of that operation would have been the equivalent of doing a rebase of develop onto master.

**Conflicts.** Conflicts can occur. If so, the user has to fix the conflicts for each of the paths in conflict (and let Git know by staging each conflicted file with add) before being able to complete (or continue) the operation. Terrible things can happen when the user is in a conflicted state. A rebase that ends in conflict, puts the user in a “detached HEAD” state. Issuing a commit in this situation will not continue the rebase but instead create work that belongs to no branch at all (the correct command is rebase --continue). To add some confusion though, if a merge results in a conflict, commit actually happens to be the correct command to execute to finalize the merge after fixing all conflicts (there is no merge --continue).

The promise of a branch being an independent line of development falls through the cracks in this state. Say the user is halfway through finishing to fix a conflict and wants to put aside this task for later and instead proceed to work on another task (say, fix a bug). Switching to another branch will not be possible, since the working directory and staging area probably have changes that would be clobbered. Even
stashing cannot help in this situation—stashing merely saves version of files, it cannot save a merge (or rebase, or cherry-pick) in progress. The user is left with no choice other than handling this out-of-band (maybe by doing a new clone of the repository and fixing the bug there, or by copying all the changes out from the repository and aborting the operation).

4.6 Syncing with other Repositories

Git allows the user to give a name to a remote repository (with the `remote` command). So, for example, if the user is constantly syncing changes with a remote repository she could give it a name (e.g., `shared`) and use it to designate this repository instead of having to use the complete URL to the repository (which might look something like `http://github.com/user/repo`—hardly something easy to remember or pleasant to type). An execution of the `clone` command (in addition to creating a local clone of the specified repository) automatically creates a remote `origin` to designate the remote repository from which the local was cloned.

Crucial to the understanding of how syncing with other repositories work is the notion of a “remote branch.” A remote branch is a branch (pointer to a commit) that (asynchronously) reflects the state of a branch in another repository. These are not (supposed to be) updated directly by the user, but instead are (usually) updated whenever there is some network communication (e.g., after an execution of `push` or `fetch`).

Fig. 4-5 illustrates how `push` and `fetch` move this remote branch (in addition to uploading or downloading any commits). In this scenario the user is working with a remote repository (designated with the name `origin` in the local repository), and there is a branch `master` both in the user’s local repository and on the remote one. In the user’s local repository there is also a remote branch `origin/master` that tracks the remote `master` branch (the one in the remote repository designated with the name `origin`). After a `push` of `master` onto the remote `master`\(^4\) (Fig. 4-5a),

\(^4\)Note that it is possible to push any branch onto any other branch in the remote repository—they
Figure 4-5: Syncing with another repository
the commits that are missing are uploaded to the remote repository, and the remote branch origin/master is set to point to the same commit the local master points to (and is also now in sync with the remote master). After a fetch of the remote master (Fig. 4-5b), the missing commits are downloaded to the local repository, and origin/master is set to point to what the remote master points to (they are now in sync). Also, a new reference FETCHHEAD appears, this (short-lived) reference is used to keep track of what has just been fetched. This reference is rarely used directly by the user, but it is implicitly used by pull (after fetch it does a merge—or rebase—of the current branch with FETCHHEAD).

While it is always possible to fetch changes, push will complain if the local branch being pushed is not ahead of the branch to which the changes are being pushed to in the remote repository. This, as expected, happens very often when working within a team where other users are pushing to the same repository. Like in CVCSs, the solution is to fetch the changes, integrate them with the local, yet-unpushed, changes, and re-push after that.\footnote{The user could also force the push, but this is rarely the correct solution.}

The notion of a “remote branch” must not be confused with that of an “upstream branch.” Git allows the user to configure a branch to have an upstream branch. This is merely for convenience (so that the user does not have to input a target branch every time): commands like pull and push default to looking at the upstream branch of the current branch to figure out where to fetch and push changes. In a normal setup, in the scenario depicted by Fig. 4-5, the local branch master would probably have origin/master set as its upstream branch. So a pull or push without any arguments would look at HEAD to figure out what the current branch is (master), see what the upstream branch of master is (origin/master), and thus look at the remote origin and its reference master for pulling/pushing changes.
4.7 History

While in CVCSs it was common to see the history of a repository as a chronologically ordered list of commits, in DVCSs the set of commits tend to form more intricate structures: it is common for several commits in the repository to have not just one but two or more parents (due to merges).

The `log` command, takes the commit pointed to by the current branch, and walks up (towards the root), reporting all commits found along the way. In a rather misleading fashion, it—at least without any argument—outputs commits one on top of the other, suggesting a chronologically ordered list of commits, which is not the case in presence of merges. (The `--graph` argument to `log` can be used to get a more faithful representation.)
Chapter 5

A Conceptual Model of Git

5.1 The Abstract State Space

5.1.1 Notation

To describe the abstract state space we use a relational data model, using the variant of extended entity-relationship diagrams developed for the Alloy modeling language [19]. This form of description has served us well so far: it is implementation independent; easy to understand yet precise enough to support objective analysis; and it is lightweight, presenting little inertia to the exploration of different points in the design space.

Instead, we might have chosen any of the well-known “model based” specification languages (such as Z [35], B [1], VDM [22], or Alloy). Our own preference is for a diagrammatic representation of the state space, but it may not be essential. State transition diagram formalisms (such as Statecharts [16]) would be less suitable, because such notations do not support richly structured state.\(^1\)

Nothing in our notation is specific to this case study, and it should be applicable

\(^1\)On the flip side, it is hard to express rich full state invariants in our notation (and more generally, in any diagrammatic notation). We plan to extend our notation to make it more expressive in the future, but are cognizant of the fact that diagrammatic syntaxes for first order logic have a long and troubled history (and that, anyway, richer invariants are easily expressed textually, for example in Alloy). For the time being, we textually state some of these invariants that are not captured in the diagram (Sec. 5.1.3).
in other domains (and we indeed use this same notation in our other case studies).

The relational data model in Fig. 5-1 show the concepts that form the abstract state of Git. Each box represents a set of atoms, and the arcs represent relations (between atoms). (An atom is an indivisible, immutable, distinguishable entity.) A large, open-headed arrow denotes a classification relationship; thus, for example, **Bare Repository** and **Non-bare Repository**, are subsets of **Repository**. Moreover, the fact that they share an arrowhead indicates that these are disjoint subsets of the set **Repository**. The bar before the arrowhead in **Repository** indicates that once a repository is a **Bare Repository** or a **Non-bare Repository**, it will always be so. In other words, a **Non-bare Repository** cannot turn into a **Bare Repository** and vice versa. The italicization of **Repository** indicates that its subsets exhaust it; that is, every repository is bare or non-bare.

A small, closed arrow denotes an association relationship (mathematically, a binary relation or set of ordered pairs). So, for example, the arc labeled `tracked` from **Non-bare Repository** to **File Path** associates a (non-bare) repository with its set of “tracked” file paths. An arc labeled `$r[i]$` represents not just one relation, but a collection of relations indexed by `$i$`. Thus the arc labeled `staged[f,p]` represents an association between (non-bare) repositories, and their staged version for each file path `$f,p$`. (Alternatively, as in Alloy, such a relation can be viewed as ternary; in this view, `staged` is a relation on (non-bare) repos, versions and file paths.)

The arc labeled `points`, with two arrowheads, associates a **Reference** with one of a **Commit** or **Reference**. It is equivalent to having two arcs, one from **Reference** to **Commit** and another one from **Reference** to itself.

The notation uses look-across cardinalities, where `!` stands for exactly one, `?` denotes zero or one, `*` (or no marking) zero or more, and `+` one or more. For arcs that represent a collection of relations, the cardinalities apply to each of the indexed relations; thus the markings on `staged[f,p]` indicate that, for each file path, a (non-bare) repository has at most one staged version.

Concepts correspond to state components of the state machine. Thus, a concept could correspond to any combination of boxes (sets of atoms) and arcs (relations).
Figure 5-1: The abstract state space of Git’s conceptual model
For example, the “staging area” is modeled by the staged and staged for removal relations, as well as the Non-bare Repository, Version and File Path sets; and the notion of a “file path classification” is modeled by the tracked, untracked, staged for removal, ignored, in conflict, and assume unchanged relations, and the Non-bare Repository and File Path sets.

5.1.2 Discussion

It is worth elaborating on the rationale of some aspects of our model. Note first that a file in Git is essentially a file path (which we attempt to make clear by using the term “File Path” instead of just “File” in the diagram). While documentation and commands like status talk about “files,” these must not be confused with the regular notion of a Unix file. In Unix, a file has a path. Renaming a file merely changes the path property of the file, but it is still the same file. If you rename a file that was committed before, and execute the status command, Git will not detect the fact that the same file changed its name, and instead show as if the file was deleted and a new one with a different name was created. Interestingly, when you add the new file then it might detect the rename. (Alternatively, users can also do git mv, which is just a shorthand for git rm followed by git add.) This is Git’s attempt to provide the illusion that it tracks files instead of paths. While it might do a good job in the simple case, if you happen to change the file significantly from its committed contents then it will not detect the rename. Symmetrically, it might detect as a rename something that is not. In another words, from Git’s perspective, mv foo bar is the same as cp foo bar; rm foo, while these are very different operations in Unix. Note that by “detecting the rename” we merely mean that the status or commit command will show that a file was renamed. Git does not store any extra information to track renames, it all depends on the heuristics each command applies to tell whether some file was renamed or not (think that commands like log or merge also need to do rename detection).

Another thing to note is that Git tracks file paths, not directories. This means that, for example, committing an empty directory is impossible. This might come as
a surprise to novices, who are used to executing the `add` command with a directory as input. It works fine when there are files under that directory, because the command interprets it as “add all files in this directory,” but nothing will happen if there is no file to track under the directory. To overcome the limitation of not being able to commit an empty directory, users resort to creating a dummy file in the directory. (Not surprisingly, in the 2011 Git User’s Survey, for the question “Which of the following features would you like to see implemented in Git?”\(^2\), the choice “support for tracking empty directories” got selected by 2045 respondents\(^3\) becoming the second most desired feature of all.)

Recall that in distributed version control, each repository has all of the version control information. This means that, in our model, the same commit (atom) could be in multiple repositories (given by the + in the Repository end of the commits arc—the commits relation could relate the same commit with multiple repositories).\(^4\) On the other hand, while most repositories have a HEAD that points to the current commit (either via a branch or directly), the reference (atom) that is acting as head is different in each repository (given by the ! in the Repository end of the head arc); even if they all point to the same commit. If they were the same reference, then resetting HEAD to point to somewhere else in one repository would also affect all the other repositories (which is not what happens in reality).

One thing that might strike one as surprising is the “at most one” cardinality of the head relation (given by the ? at the Reference end of the head arc). Can a repository have no HEAD at all? This is indeed possible: a repository that has no commits, has nothing for HEAD to point to, and has no HEAD at all. This actually happens to be a problem for many users, who after creating a remote repository want to clone it immediately to their own machine, but eventually realize that the repository has to have at least one commit in order to be possible to clone it; which


\(^3\)33% of the total.

\(^4\)It is the unmutability of a commit that allows us to model it in this way and still faithfully represent what happens in reality: as soon as some aspect of a commit changes (e.g, its contents, or its parents), the SHA (id) changes, and thus becomes a new commit.
is one of the reasons why when you create a new repository in GitHub, they prompt (and encourage) you to create a new commit (with a README and/or LICENSE file).

5.1.3 Other Constraints

Diagrams such as the one used in Fig. 5-1 do not express the full state invariants. Some of the constraints not captured are about maintaining the local/remote boundary. For example, that the parent(s) commit(s) of a commit must belong to the same repository it belongs; or that a reference can only point to a reference or commit that belongs to the same repository it belongs; or that the upstream of a branch must belong to the same repository it belongs.

Other constraints (and perhaps the most interesting ones) are related to file path classifications: for example, that a staged for removal file path has no staged version; or that a tracked path has a version at the last commit point while an untracked does not. Actually, what constitutes a tracked or untracked path is more complicated than that—but more on this later in the next section.

5.2 Explanations

Table 5.1 shows for each concept its definition, and Table 5.2 shows its purpose. Together, these constitute the explanations for Git’s concepts. The explanations are based on popular Git references. Some explanations are more challenging than it might appear to be at first sight, and worthy of a discussion (which we include below).

5.2.1 Tracked and Untracked

Definition. Roughly, a tracked path is a path that has a committed version in the current commit point (i.e., the one pointed to by HEAD), and, conversely, an untracked path is one that has no committed version in the current commit point. Unfortunately, the intrusion of the staging area makes (accurately) defining these two notions very
<table>
<thead>
<tr>
<th>Concept</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Repositories</strong></td>
<td></td>
</tr>
<tr>
<td>Bare Repository</td>
<td>Repository that contains the version control information but has no working directory</td>
</tr>
<tr>
<td>Non-bare Repository</td>
<td>Repository that contains the version control information and has a working directory</td>
</tr>
<tr>
<td>Commit</td>
<td>Complete snapshot of the repository at some point in time with a comment (that summarizes the changes), timestamp, and author information</td>
</tr>
<tr>
<td>Working Directory</td>
<td>Mapping of the repository contents to the local hard drive</td>
</tr>
<tr>
<td>Staging Area</td>
<td>Area that stores the collection of file versions which a subsequent commit would save (it stores information about what will go into the next commit)</td>
</tr>
<tr>
<td>Stash</td>
<td>Collection of (working) file versions that are saved together. Stashing takes the dirty state of the working directory, saves it, and resets the working directory to its state at HEAD. The saved stash can be later reapplied onto the working directory</td>
</tr>
<tr>
<td><strong>References</strong></td>
<td></td>
</tr>
<tr>
<td>Head</td>
<td>Pointer to the current commit</td>
</tr>
<tr>
<td>Tag</td>
<td>Static, named pointer to a commit</td>
</tr>
<tr>
<td>Branch</td>
<td>Movable, named pointer to a commit</td>
</tr>
<tr>
<td>Upstream Branch</td>
<td>Branch to which operations like merge, rebase, push, pull default to</td>
</tr>
<tr>
<td>Remote Branch</td>
<td>Branch that asynchronously reflects the state of a branch in another repository</td>
</tr>
<tr>
<td><strong>File Path Classifications</strong></td>
<td></td>
</tr>
<tr>
<td>Tracked</td>
<td>Roughly, a path that has a committed version in the current commit point, but see Sec. 5.2.1</td>
</tr>
<tr>
<td>Untracked</td>
<td>Roughly, a path that has no committed version in the current commit point, but see Sec. 5.2.1</td>
</tr>
<tr>
<td>Ignored</td>
<td>Path that is completely ignored by Git, will not show in status</td>
</tr>
<tr>
<td>Assume Unchanged</td>
<td>Path that was previously tracked, but is now ignored</td>
</tr>
<tr>
<td>In Conflict</td>
<td>Path which is in a conflicted state. It requires some manual intervention to merge changes</td>
</tr>
</tbody>
</table>

Table 5.1: Definitions for Git's concepts
<table>
<thead>
<tr>
<th>Concept</th>
<th>Motivating Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Repositories</strong></td>
<td></td>
</tr>
<tr>
<td>Bare Repository</td>
<td>Hub to upload/download changes from</td>
</tr>
<tr>
<td>Non-bare Repository</td>
<td>Do work in disconnected mode</td>
</tr>
<tr>
<td>Commit</td>
<td>• Make set of changes to files persistent</td>
</tr>
<tr>
<td></td>
<td>• Group logically related changes together</td>
</tr>
<tr>
<td>Working Directory</td>
<td>Provide a platform for users to edit files</td>
</tr>
<tr>
<td>Staging Area</td>
<td>Select changes that should go into the next commit</td>
</tr>
<tr>
<td>Stash</td>
<td>Clean up the working directory of changes, while saving them so that they can be later reapplied</td>
</tr>
<tr>
<td><strong>References</strong></td>
<td></td>
</tr>
<tr>
<td>Head</td>
<td>Easy way of naming the current commit</td>
</tr>
<tr>
<td>Tag</td>
<td>Signify that a commit is special in some way</td>
</tr>
<tr>
<td>Branch</td>
<td>Switch between parallel lines</td>
</tr>
<tr>
<td>Upstream Branch</td>
<td>Set a default for sync operations</td>
</tr>
<tr>
<td>Remote Branch</td>
<td>Do work in disconnected mode</td>
</tr>
<tr>
<td><strong>File Path Classifications</strong></td>
<td></td>
</tr>
<tr>
<td>Tracked</td>
<td>Roughly, to mark files which have a committed version in the current commit point, but see Sec. 5.2.1</td>
</tr>
<tr>
<td>Untracked</td>
<td>Roughly, to mark files which do not have a committed version in the current commit point, but see Sec. 5.2.1</td>
</tr>
<tr>
<td>Ignored</td>
<td>Mark files which are to be completely ignored (will thus not show in status or go into a commit)</td>
</tr>
<tr>
<td>Assume Unchanged</td>
<td>Mark files which are to be completely ignored (will thus not show in status or go into a commit)</td>
</tr>
<tr>
<td>In Conflict</td>
<td>Mark files which have conflicts that need to be resolved manually by the user</td>
</tr>
</tbody>
</table>

Table 5.2: Motivating purposes for Git’s concepts
challenging. Apparently, even popular Git references are confused on what a tracked and untracked file mean.

For example, the Pro Git book [8] describes tracked files as “files that were in the last snapshot; they can be unmodified, modified or staged.”\(^5\) It is not entirely clear what “files that were in the last snapshot” means, but from the context it appears to mean “files that were in the current commit point.” The book then says “Untracked files are everything else,” but then qualifies this, surprisingly, with “[namely] any files in your working directory that were not in your last snapshot and are not in your staging area” suggesting that tracked files also include files that have been staged but are not part of the last snapshot, contradicting the earlier definition.

Even this definition of untracked file is not consistent with Git’s behavior. For example, executing `rm --cached` of a tracked path stages the removal of the path without touching the file in the working directory, so that it will be deleted from the repository at the next commit. Subsequently running `status` will show the file under both the “Changes to be committed” section (as “deleted”) and in the “Untracked files” section. So the same path can effectively be both a staged for removal and untracked file path. We thus now have an untracked file that has a version on the current commit point!

**Purpose.** In general, the motivating purpose of having file path classifications is clear: it is to classify paths into groups where each group has some special semantics. Then, instead of naming each path individually one could use the group to (succinctly) refer to all paths that belong to it, or actions could default to taking as input one of these groups.

The question would then be whether each group has a compelling purpose for its existence. The purpose of “In Conflict,” “Ignored,” and “Assume Unchanged” are clear (as shown in Table 5.2); though one could question why have two concepts, “Ignored” and “Assume Unchanged,” that seem to have the same purpose—more on this later in the analysis chapter.

\(^5\)See Chapter 2.2, “Recording Changes to the Repository.”
Since the precise definition of tracked and untracked is unclear, so is their motivating purpose. Roughly, it seems to be to classify files into whether they have a committed version in the current commit point or not.

5.3 Purpose Graph

The last thing we are missing for our model of Git to be complete is to identify which purposes are subpurposes of others. A purpose is a subpurpose of another purpose if it supplements it. For example, “signify a commit is special in some way” supplements “represent/record coherent points.” Fig. 5-2 shows the purpose graph. The high level purposes for version control we identified before appear at the top of the graph. An arrow denotes a “subpurpose” relationship; thus, for example, “select changes that should go into the next commit” is a subpurpose of “group logically related changes together.”

5.3.1 Stash

The motivating purpose of stashing (“clean up the working directory of changes, while saving them so that they can be later reapplied”) appears to be not connected to any high level purpose. Why is that so?

Take (what seems to be) the motivating use cases for stashing (which are also mentioned in the command documentation\(^6\), and book\(^7\)): pulling into a dirty working directory and dealing with an interruption in your workflow (i.e., you are in the process of working on something, say a feature or bug fix, and now suddenly need to work on something else).

The first refers to the case where you have some (uncommitted) changes in the working directory, which happen to conflict with the changes that are being pulled. In this case, the pull command will (very sensibly) refuse to overwrite the changes in your working directory and abort (i.e., the pull will \textit{not} happen). The workaround

\(^7\)See [8], Chapter 6.3.
Figure 5-2: Git's purpose graph
(assuming you want to keep the changes) is to create a stash (which will clean the working directory), do the pull (which will now succeed since there are no changes in the working directory), and then reapply the stash to your working directory, and fix conflicts.

The second refers to the case where the (uncommitted) changes in the working directory prevent you from being able to do some other (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). This also captures the case in which you have to switch to a different branch but are unable to do so because the changes in the working directory would conflict with the changes in the destination branch.

In both the aforementioned scenarios, there are other alternatives for reaching the same final state, but they involve more work. What makes stashing so compelling is the fact that is a quick and easy way of cleaning the working directory (thus leaving you in a state where it is possible to pull conflicting changes, do other work in the same branch, or switch branches) while saving the uncommitted changes.

Since stashing is persisting a set of changes, one could map the purpose of stashing to “making a set of changes persistent,” but this does not seem to be its (high level) motivating purpose as the use cases evidenced. There is more in stashing going on that just saving changes. 8

How about “switching between parallel lines”? It is, after all, in some sense, letting you switch between two (implicit) lines of development. One that has your local (uncommitted) changes, and the other one that consists of the new commits that are to be pulled (in the first use case), or of the new changes to be developed (in the second use case). But this connection is tenuous.

In conclusion, we believe stashing’s purpose is not a subpurpose of any high level purpose.

---

8 This is even made more clear by looking at the opening sentence in the description of the stash command in Git’s documentation: “Use git stash when you want to record the current state of the working directory and the index, but want to go back to a clean working directory. The command saves your local modifications away and reverts the working directory to match the HEAD commit.”
Chapter 6

Analysis of Git’s Conceptual Model

The very complexity of the conceptual model we have described suggests to us that something is amiss in the design of Git. Can a version control system really require so many, and such intricate concepts? In this section, we attempt to point to some particular problems, namely examples of ways in which Git violates the criteria we outlined in Sec. 2.1.

6.1 Commit

The notion of a commit is pervasive in version control; all of the popular VCSs have it. The problem with commit is that it constitutes a violation to the compelling criteria: the same concept (commit) has more than one, unrelated, purpose. In the case of commit, these are to “make a set of changes to files persistent” and “group logically related changes together.” In Chapter 3 we pointed out how these two purposes belong to different categories: “data management” and “change management” respectively.

An interesting thing to note is that these two purposes are not only unrelated, but also in tension with each other. On one hand, you would like to save your changes as often as possible, so that if something happens you loose as few data as possible (thus making as many commits as possible). On the other hand, a logically related group of changes usually involve multiple individual changes, which means you might be working for quite some time before you decide to group a set of changes (thus
waiting to make a commit until you have sufficient changes to group together).

Those users that are very strict about just using commit as a change grouping tool search for different ways to fulfill the purpose of saving changes. For example, they put their Git repositories in Dropbox (which comes with its share of problems—it is not uncommon for users to get their repositories corrupted). Others tend to commit often and then worry about doing the real grouping of changes before these are shared with the outside world (i.e., before doing a push or submitting a “pull request”). So they save changes to the repository multiple times, and then worry about grouping changes together. By “cleaning up history” (using rebase), they can regroup all the changes made in a different (more compelling) way. In some sense, they achieve less “persistence” than those that go for the Dropbox solution (since the data is only saved in their local repository it might not survive hardware failure, for example), but they do not need to deal with other tool other than Git, and it does give some fault tolerance. In both cases, it is a lot of extra work the users have to do to work around the problem with commit.

6.2 File

We previously talked about the difference between Git’s concept of “file,” which is essentially a file path (thus the use of the term “File Path” in our model to make it more clear), and the more common understanding of a Unix file, where the path is merely a property of the file (Chapter 5). This inconsistency constitutes a violation of our conventional criteria. Simply reusing the existing concept of file would have avoided some surprises to users that are accustomed to the regular concept of file.

6.3 File Path Classifications

The problem with Git’s choice of file path classifications is that they are not controllable. Git distinguishes files according to whether they are in the current commit or not, whether they are staged or not, and whether they are matched by a ignore spec
or not. Leaving aside the complexity due to the myriad of potential states a file could be in, the main problem is that is very difficult to control the class to which a file belongs. Making an untracked file tracked is easy; you stage it. Same if you want to make an untracked file ignored; you modify the ignore spec so that the file is matched by it. But what if you want to untrack or ignore a tracked file? This is impossible; the only thing you can do is mark it as “assume unchanged,” which will make the file behave, for the most part, very similar to an ignored file.

6.4 Staging Area

The staging area is praised by some, reviled by others. Git boasts this concept as a unique advantage of Git with respect to other VCSs; yet for most users it offers inessential functionality, and its presence complicates the use of Git considerably.

We believe the main problem with the staging area is that it is not controllable; it interferes with too many other concepts. The consequence of this is that a user of Git simply cannot ignore the staging area; it is a piece of the software that needs to be learned (thus steepening the learning curve). The staging area is so entrenched in Git’s core design that it permeates everywhere. How do you create a new commit? By first staging files (either explicitly with the add command or implicitly by passing flags to the commit which do an implicit add). What’s a tracked file? Well, it is a file that was in the last commit, or staged. How do you let Git know that the conflicts in a file have been resolved? By staging the conflicted files. The first shows the interference of the staging area with commit, the last two, its interference with file path classifications.

Interference with Commit. The output of status evidences the interference of the staging area with commit. For example, after modifying a tracked file, the file will appear under the “changes not staged for commit” section. That’s a clear reference to the staging area. Moreover, the instructions in the output suggest to use add to update what will be committed. Clearly, the default route for creating a commit is to
explicitly make use of the staging area. Even if in most cases the commits are fairly simple, and thus wouldn’t require this intermediate step.

The Pro Git Book [8] seems to acknowledge that the staging area is sometimes “a bit more complex than you need in your workflow” and suggests using the -a option of the git commit command, which automatically stages every file that is tracked and modified, to “skip” the staging area.\(^1\) This does not skip the staging area; it merely lets you avoid having to do an (explicit) add (files will still flow through the staging area). This option is mostly intended to enable faster workflows, not to hide the existence of the staging area. For example, to commit files unknown to Git (“untracked” files), these have to be explicitly staged first (with the add command).

**Interference with File Path Classifications.** The fact that to start tracking a file one has to stage it points to another interference: one with file path classifications. The action of staging, not only adds the changes to the file to the staging area, but also classifies the file path as “tracked.” And this is not the only example of interference with file path classifications: staging is also used to mark a file with conflicts as resolved, thus removing the file from the “in conflict” class.

### 6.5 Branch

The problem with Git’s concept of a branch is that it is not compelling: it is motivated by one purpose (switching between parallel lines) but it has a rather complex operational principle.

While branches let users have multiple parallel lines of development, there is only one working directory and staging area; so any uncommitted changes will stay after the switch. This is potentially problematic: changes could conflict with what is on the destination branch, which prevents the user from switching\(^2\); and, even if there are no conflicts, the changes (probably) do not belong to this new branch (because, presumably, they are unrelated to what the user needs to do next).

\(^1\)See [8], Chapter 2.2.

\(^2\)Git fails with an error if there are any conflicting changes.
To solve this problem, users resort to stashing, or doing an incomplete commit (which will have to be amended afterwards). This is an extra set of actions the user needs to perform, and more concepts and features to learn about, in order to do something as mundane as switching to a new branch.

But this is not the end of the story. If the user is in the middle of fixing conflicts, then it is simply not possible to switch to another branch. Stashing will not work: while it is possible to stash versions of files to be later retrieved, the user cannot stash a merge (or rebase or cherry-pick) in progress. The options are to either abort the operation and loose work, or continue with it so as to finalize it and later amend all the spurious commits.

Another alternative is to keep as many separate clones of the repository as needed. (This is actually what Mercurial recommends if the user wants independent working directories.) But this introduces other problems, such as, keeping the multiple local repositories in sync, and switching between branches (which could be more disruptive since the user needs to move to another directory).³

6.6 Stash

The problem with the concept of stash is that it is not compelling. While (as Table 5.2 shows) it does have a motivating purpose (to “clean up the working directory of changes, while saving them so that they can be later reapplied”), this does not map to any of the high level purposes we identified (as the purpose graph, Fig.5-2 shows). We believe there is no clear mapping to a high level purpose because the motivating purpose for the inclusion of stash in Git results from other decisions in Git’s design, and not from the intrinsic complexity of version control. In other words, maybe if branching would work differently, there wouldn’t be a need to stash changes before switching to a new branch, and the same with syncing with other repositories. (More on this in the Gitless chapter.)

³Not to mention the increase in space usage—but this is rarely an issue unless the repository is very big.
Chapter 7

Gitless

Gitless\(^1\) (our VCS) is an experiment, not a demonstration and validation of our philosophy of conceptual design. Moreover, we do not claim that Gitless occupies an ideal point in the design space. Of course we hope that a redesign of Git that “improves” (according to our criteria) its conceptual model will result in a better and more usable VCS, but we regard this as a research hypothesis to be tested empirically.

Gitless has evolved with time (and continues to evolve); the discussion here corresponds to version 1.0. As of this writing, most aspects of this design have already been implemented, but a few features remain.

Our implementation acts as a veneer on Git that changes the underlying concepts. Gitless is compatible with Git: every Git repository is also a Gitless repository (and vice versa). This way of implementing Gitless was our choice by design: it makes adoption by existing Git users frictionless, and we do not have to worry about low-level details of implementing a VCS.

On the flip side, it is also a limiting factor: Git is our canvas. This impairs our ability to implement an artifact that embodies a more radical point in the concept design space. Yet, we believe Gitless represents an interesting point in the concept design space, and an interesting exercise in conceptual redesign of software.

We do plan to develop and experiment with other, more radical, solutions to version control (in general, not specific to fixing problems in Git) in the future—more

\(^1\)http://gitless.com.
on this in Chapter 9.

7.1 Overview

Gitless has no staging area, and there are only the following file path classifications: “tracked,” “untracked,” “ignored,” and “in conflict.” A tracked file is a file whose changes will be tracked by Gitless (and automatically considered for commit); an untracked file is one whose changes will not be tracked by Gitless; an ignored file is a file which is completely ignored (will not show in status); and a file in conflict is one which is in a conflicted state (and requires some manual intervention to resolve conflicts).

Files can move freely between these groups: track makes a file tracked, untrack makes it untracked, the ignore mechanism works in the same way as Git’s (with an ignore spec given by the “.gitignore” files), conflict and resolve mark a file as having conflicts or not. It does not matter whether the file has a version at the current commit point or not; a file with a version in the current commit point can be made untracked, ignored, and it can even be marked as having conflicts.

In its default form (with no arguments given), the commit command commits the working version of all tracked files with modifications. But the set of files to commit can be further customized by either: (i) explicitly specifying a set of files; if so, only these files will be committed; (ii) specifying a set of tracked files to exclude via the -exc flag; or (iii) specifying a set of untracked files to include via the -inc flag.

A branch in Gitless is not merely a (movable) pointer to a commit but 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 path 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 that end up in conflict, the user is still working on the original branch, and can switch to another branch and then go back to fixing the 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).

Gitless has no (explicit) notion of a “remote branch” (a local branch that asyn-
chronously reflects the state of a branch in another repository). This is hidden from
the user; there is no fetch command. Whenever the user performs some command
that needs information about a branch in another repository, Gitless will automati-
cally do the fetch (or push).

7.2 Conceptual Model

The concepts that form the abstract state of Gitless are shown in the relational data
model of Fig. 7-1; Table 7.1 has the definitions for Gitless’s concepts and Table 7.2
has the purposes for Gitless’s concepts (together, these constitute the explanations
for Gitless); and Fig. 7-2 shows the purpose graph.

7.3 Gitless versus Git

The key differences between Git and Gitless’s model are:

- the elimination of the staging area;

- the redefinition of file path classifications, and the elimination of “assume un-
  changed”;

- the redesignation of the concept of “branch” as a truly independent line of
  parallel development: the creation of a “current” pointer to the current/active
  branch in a repository, indexing of working versions by branch, and the re-
  designation of HEAD to be a per-branch pointer to the current commit in the
  current branch;

- the elimination of the concept of “stash” and “stashed” versions;

- and the elimination of the explicit notion of “remote branch.”
Figure 7-1: The abstract state space of Gitless's conceptual model
<table>
<thead>
<tr>
<th>Concept</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Repositories</td>
<td></td>
</tr>
<tr>
<td>Bare Repository</td>
<td>Repository that contains the version control information but has no working directory</td>
</tr>
<tr>
<td>Non-bare Repository</td>
<td>Repository that contains the version control information and has a working directory</td>
</tr>
<tr>
<td>Commit</td>
<td>Complete snapshot of the repository at some point in time with a comment (that summarizes the changes), timestamp, and author information</td>
</tr>
<tr>
<td>Working Directory</td>
<td>Mapping of the repository contents to the local hard drive</td>
</tr>
<tr>
<td>Head</td>
<td>Pointer to the current commit</td>
</tr>
<tr>
<td>Tag</td>
<td>Static, named pointer to a commit</td>
</tr>
<tr>
<td>Branch</td>
<td>Independent line of development</td>
</tr>
<tr>
<td>Upstream Branch</td>
<td>Branch to which operations like <code>merge</code>, <code>rebase</code>, <code>publish</code> default to</td>
</tr>
<tr>
<td>File Path Classifications</td>
<td></td>
</tr>
<tr>
<td>Tracked</td>
<td>Path whose changes will be tracked by Gitless (will show in <code>status</code>, and be automatically considered for commit)</td>
</tr>
<tr>
<td>Untracked</td>
<td>Path whose changes will not be tracked (will still show in <code>status</code>, but it will not be automatically considered for commit)</td>
</tr>
<tr>
<td>Ignored</td>
<td>Path that is completely ignored by Git, will not show in <code>status</code></td>
</tr>
<tr>
<td>In Conflict</td>
<td>Path which is in a conflicted state. It requires some manual intervention to merge changes</td>
</tr>
</tbody>
</table>

Table 7.1: Definitions for Gitless's concepts
<table>
<thead>
<tr>
<th>Concept</th>
<th>Motivating Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><strong>What is it for?</strong></td>
</tr>
<tr>
<td><strong>Repositories</strong></td>
<td></td>
</tr>
<tr>
<td>Bare Repository</td>
<td>Hub to upload/download changes from</td>
</tr>
<tr>
<td>Non-bare Repository</td>
<td>Do work in disconnected mode</td>
</tr>
<tr>
<td>Commit</td>
<td>• Make set of changes to files persistent</td>
</tr>
<tr>
<td></td>
<td>• Group logically related changes together</td>
</tr>
<tr>
<td>Working Directory</td>
<td>Provide a platform for users to edit files</td>
</tr>
<tr>
<td>Head</td>
<td>Easy way of naming the current commit</td>
</tr>
<tr>
<td>Tag</td>
<td>Signify that a commit is special in some way</td>
</tr>
<tr>
<td>Branch</td>
<td>Switch between parallel lines</td>
</tr>
<tr>
<td>Upstream Branch</td>
<td>Set a default for sync operations</td>
</tr>
<tr>
<td><strong>File Path Classifications</strong></td>
<td></td>
</tr>
<tr>
<td>Tracked</td>
<td>Mark files whose changes will be automatically tracked</td>
</tr>
<tr>
<td>Untracked</td>
<td>Mark files whose changes will not be tracked</td>
</tr>
<tr>
<td>Ignored</td>
<td>Mark files which are to be completely ignored (will thus not show in <code>status</code> or go into a commit)</td>
</tr>
<tr>
<td>In Conflict</td>
<td>Mark files which have conflicts that need to be resolved manually by the user</td>
</tr>
</tbody>
</table>

Table 7.2: Motivating purposes for Gitless’s concepts
Figure 7-2: Gitless's purpose graph
7.3.1 A Compelling Concept of Branching

From a workflow perspective, switching branches in Gitless is (roughly) equivalent to always creating a stash before switching to a different branch in Git, and then retrieving this stash when the user switches back to the original branch. (In reality, there is a bit more going on because we also maintain the classification markings per branch.) Thus, it is as if there are multiple working directories (one for each branch), or in other words, one can think of it as a file potentially having several working versions accessible via a branch name $b$ (noted as working$[b]$ in the diagram). This means that the user can freely switch from branch to branch without having to stash or commit unfinished changes. We believe this lives up to the expectation of a branch being an independent line of development.

Also, a user in Gitless is always working on some branch (even during operations that lead to conflicts that need to be manually resolved), i.e., it is not possible to go into a “detached HEAD” state.

To address situations in which you want the changes made in the current branch to be moved onto the destination branch (for example, you might realize after some time that you are doing work in the wrong branch), the gl branch command has a --move-over flag that does this.

Another thing to note is that if the user has any uncommitted changes and attempts to do any sync operation (which would clobber the uncommitted changes) the uncommitted changes in the working directory are saved and later reapplied after the operation finishes.

Our concept of branch (and mechanics of the sync operations) significantly eliminates the motivation of stashing, which is why there is no stashing 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.
7.3.2 Controllable File Path Classifications

Our redefinition of the file path 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).

7.3.3 No Interfering Staging Area

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.

In our experience with Git, we did use staging to specify which files to commit. In this case, however, staging is compensating for the rigidity of the commit command. We address this issue in Gitless by providing a more flexible commit command (with -inc and -exc flags, and allowing untracked files to be given as input).

Another justification of the staging area is that it enables segments of files, rather than entire files, to be committed. Despite the fact that we believe this to be a less common use case, it is occasionally useful, so we address this with a -p flag in gl commit (similar to Git’s commit -p) that allows the user to interactively select segments of files to include in the commit (but without requiring staging).

Admittedly, there are some use cases for which a staging area comes in handy. For example, with Git, when doing a commit that includes several files, we have found ourselves executing git diff f; if the file f looks suitable for committing, we then execute git add f, repeating this process for each file f that has been modified, until all files have been “cleared” for commit, and then finally issuing a git commit. In this scenario, the staging area acts as a record of which files were already cleared for commit and which ones are pending. Gitless does not support this; you could untrack all the files and make them tracked one at a time, but Gitless would not maintain for you a record of the entire set of files that should eventually be tracked.

At this point, we have no (explicit) staging area in Gitless. One could still add a staging area that does not interfere with committing or with file path classifications. For example, simply by adding a gl stage command, whose only effect would be to
stage the contents of the file (and a `gl unstage` to undo). Then, if there is anything staged in the staging area, `commit` would commit those versions (instead of taking them from the working directory).

7.4 Gitless versus Mercurial

It is worth discussing some differences (and similarities) between Gitless and Mercurial\(^2\), which is the next most popular DVCS after Git (and often regarded as easier to use than Git).

Mercurial has no staging area, which greatly reduces the complexity related to recording changes and brings it closer to Gitless’s design (where there is only tracked, untracked, ignored and in conflict file paths). Yet these groups are not as controllable as in Gitless. For example, changing the status of a file from tracked to untracked can be done using the `forget` command (which works as the opposite of `add`) but only as long as the file was not committed before. Ignoring or untracking a committed file seems to be impossible (at least without using an extension).

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. From a workflow perspective, their notion of branch is not different from Git’s; there is only one working directory, so switching branches is a complex task with uncommitted changes. Worse, unlike in Git, in Mercurial there is no way to save uncommitted changes (like stash), leaving the user with the only option of committing unfinished changes before switching to a new branch (unless the user activates an extension that adds this feature\(^3\)).

In addition to the concept of “branch,” Mercurial also has something called a “bookmark.” Bookmarks are pointers to commits, equivalent to Git’s branches. They can be synced with other repositories as well. The general guideline is to use branches (also called “named branches”) for long-lived lines of development, and bookmarks for

\(^2\)[http://mercurial.selenic.com].

\(^3\)[See ShelveExtension [http://mercurial.selenic.com/wiki/ShelveExtension].]
short-term lines of development (like for bug fixes, or for the development of some specific feature). Each changeset records the branch in which it was made, but not the bookmark.

The recommended way for getting an independent working directory (like what you get with Gitless) is to create separate clones of the repository (commands like `hg pull` let you pull changes from another directory).

Mercurial has a concept of tag as well, which is the same as Gitless’s (and Git’s), defined in [30] as a “symbolic name” for a revision. One of these tags is a special, so-called “floating” tag, named “tip” which identifies the newest revision in the repository (it serves the role of `HEAD`). Interestingly, there is a file `.hgtags` which 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. All the commands that list, create, edit or delete tags are merely wrappers that modify this special file. This is certainly different from how Gitless (and Git) handle tags. We believe Git’s way of handling tags (which we copy in Gitless) is better. In Mercurial, tagging interferes with commit (each new creation/deletion of tag is done through a new commit).

---

4Not to be confused with Mercurial’s definition of a “head” which is a changeset with no child. So, for example, tip would point to a head.
Chapter 8

Related Work

8.1 Usability and Conceptual Models

The idea that design for usability should go beyond the user interface is widely accepted. As Bruce Tognazzini puts it, in his popular *First Principles of Interaction Design* [36]: “The great efficiency breakthroughs in software are to be found in the fundamental architecture of the system, not in the surface design of the interface.” For a long time, designers have felt slighted by the insinuation that they are decorators whose job is just to make a product pretty; a design should be developed “from the inside out,” according to Herbert Dreyfuss, in his classic industrial design memoir *Designing for People* [10], in which he gives many examples of assignments he rejected because the client expected him to work only on the surface appearance.

Most usability experts seem to recognize the importance of a product’s conceptual model, and the problems that arise when the user’s mental model, and the correct conceptual model of the product diverge. In his influential book [28], Donald Norman mentions issues arising from conceptual models many times, and he suggests that the designer pay attention to crafting a “system image” that reflects the conceptual model of the design. But like many writers on usability, he has little to say about the construction and analysis of the conceptual model itself, except that it should be “functional, learnable and usable.”
8.2 Conceptual Integrity

Fred Brooks put the term “conceptual integrity” on everybody’s lips when he described it in *The Mythical Man Month* [4] as the “most important consideration in system design.” Unfortunately, that book had little to say about what the term actually meant. The focus in *The Mythical Man Month* is on software process, and on Brooks’s contention that conceptual integrity requires a single designer—put most bluntly in his recent book *The Design of Design* [6] in a section entitled “Conceptual design, especially, must not be collaborative.”

It is not clear what “conceptual design” means in this context, but he seems to use it the way traditional designers do to refer to the initial and most high-level design steps. The underlying assumption seems to be that great designs emerge from single minds; Dick Gabriel has taken Brooks to task on this, contending that he misconstrues his own example of the history of development of the dome of the Florence cathedral [14].

Brooks never really defines the term “conceptual integrity.” The closest he comes is the listing of its three key principles, which first appeared in his coauthored book on computer architecture [7], and which are recapitulated in *The Design of Design* [6]. The principles are: orthogonality—that individual features should be independent of one another; propriety—that a product should have only the functions essential to its purpose and no more; and generality—that a single function should be usable in many ways. Propriety might also be called unity of purpose, most memorably articulated by a sketch of that name by the British comedians Mitchell and Webb, in which they trade examples of their frustration with products that include inessential functions. Complaining about the inclusion of a heater in his car, one proclaims: “A car is a means of transport, not a sitting room on wheels!”

A different definition of the term, found in a wiki post by Bill Griswold [15], suggests that conceptual integrity means that wherever you look in a system, you see evidence of the same overall design (or perhaps, of the same designer at work). This might better be called stylistic uniformity; and indeed, Brooks says that conceptual
integrity is called “coherence, sometimes consistency, sometimes uniformity of style” in [6]. This seems a better match to Brooks’s view that collaboration runs counter to conceptual integrity, since harmonizing the style of multiple designers is not easy. But it contradicts his definition in terms of the three key principles, which are surely orthogonal to stylistic uniformity. And, surprisingly, in a later chapter of The Design of Design [6], Brooks himself later identifies style, quoting Webster’s dictionary, as being more about “form or expression” than about “substance.”

In his widely read “The Rise of the Worse is Better” [12], Dick Gabriel presents a dichotomy between two styles of software development. One, typified by LISP, values “doing the right thing,” and never sacrifices simplicity or correctness for any other quality. The other typified by C and Unix, values growing a system piecemeal, worrying less about getting it right, and emphasizing simplicity in the implementation over simplicity in the user interface. Gabriel himself remains undecided on which approach is better [13], and has written subsequent articles on both sides. From our perspective, we see “The Rise of the Worse is Better” as a warning against the risk of attempting perfection at the expense of the many pragmatic demands of a system development. But at the same time, our critique of Git arises from the conviction that worse really is worse, and that Git’s design amply demonstrates this.

8.3 Conceptual Modeling

An entire field is devoted to “conceptual modeling” (see, for example, the textbook by Olivé [29]); it grew out of the need to find ways to describe the structure of a system’s data without making any commitments to how it is represented, originating with the entity relationship diagram [9], continuing with research on more expressive “semantic models” [18], and then merging with the development of notations for more general software design, such as the object model of OMT [34], which became the class diagram of UML [3]. Arguably, the field of formal specification had the same motivation in mind—the Z language [35] in particular grew out of Jean Raymond Abrial’s work on databases—even though emphasis was sometimes placed more on
the transitions between states than on the structure of the states themselves. The
Alloy modeling language [19] placed more emphasis on the description of the data
structure than its predecessors, and was designed to make it easier to express con-
ceptual data models textually (principally by providing a very expressive declaration
syntax to support subtyping). Also, Alloy was designed in the context of an advo-
cacy of "lightweight formal methods" [21], which emphasized capturing the essence of
a system over detailing all of its behaviors.

Bjørner's work (see, e.g., [2]) on domain models of application areas (such as
railways and oil pipelines) can be seen as a form of conceptual modeling. The aim is to
articulate the key concepts in the problem domain independently of the specification
of any particular system to be built in that domain. Conceptual model patterns play
a similar role, as found, for example, in Martin Fowler's book Analysis Patterns [11]
and in David Hay's Data Model Patterns: Conventions of Thought [17].

8.4 Conceptual Design

Despite all this work on representing concepts, much less attention has been paid
to the question of where concepts come from, whether discovered in the problem
domain or invented by the designer. In early object-oriented methods, it was com-
monly argued that the objects comprising the system emerged almost trivially from
the problem domain. Thus Bertrand Meyer in Object-Oriented Software Construc-
tion [26]: "This is why object-oriented designers usually do not spend their time in
academic discussions of methods to find the objects: in the physical or abstract re-
ality being modeled, the objects are just there for the picking!" Similarly, in John
Guttag and Barbara Liskov's Abstraction and Specification in Program Development
[25], their method entails picking "abstractions" from the requirements specification,
which are then elaborated using "helper abstractions" into the program structure;
where the original abstractions come from is not explained. (A later edition of the
book [24], incidentally, introduced conceptual models for requirements.)
Chapter 9

Future Work

In the near future, we plan to finish and refine our implementation of Gitless. The goal is for Gitless to be not merely a research prototype but a real tool people can rely on for their version control. We also plan to develop and experiment with more radical solutions to the problem of version control.

Over the long term, we have the goal of building a rigorous foundation for concept design. This will include developing notations for capturing key conceptual issues, extending and refining the criteria, and applying them in more diverse case studies.
Bibliography


76


