What's Wrong with Git? A Conceptual Design Analysis

Santiago Perez De Rosso Daniel Jackson

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

Abstract

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 paper reports on the early stages of our first study, on the Git version control system. Despite its widespread adoption, Git puzzles even experienced developers and is not regarded as easy to use. In an attempt to understand the root causes of its complexity, we analyze its conceptual model and identify some undesirable properties; we then propose a reworking of the conceptual model that forms the basis of (the first version of) Gitless, an ongoing effort to redesign Git and experiment with the effects of conceptual simplifications.

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

Keywords concepts; concept design; conceptual integrity; conceptual modeling; design; software design; usability; version control; Git.

1. Introduction

Background and Motivation. In many areas of software development, researchers have recognized the importance of identifying underlying concepts, and of separating them from their realization in code. Doing so enables conceptual design to be pursued independently of implementation decisions; in short, the most basic form of "what" before "how."

Nevertheless, most research to date has focused narrowly on the question of how to represent concepts – as database semantic models, formal specifications, ontologies, and so on. Much less attention has been paid to the concepts themselves: how they are chosen, and the role they play in design.

Sometimes the conceptual basis of a software product is built by analogy to the real world; an online store, for example, has concepts such as "shopping cart," "item" and "order." But often the connection to the real world is tenuous and concepts make sense only in the designer's invented world. Git is an example of this, with concepts such as "tracked file," "staging area" and "local repository" that have no *a priori* meaning.

The best software designers already know how important it is to discover or invent the right concepts, and that rough edges in these concepts and their relationships will lead to rough edges in the delivered product. Back in 1975, in *The Mythical Man Month* [13], Fred Brooks described conceptual integrity as "the most important consideration in system design" and twenty years later in the afterword of a new edition [14] wrote "I am more convinced than ever. Conceptual integrity is central to product quality."

Despite this, there have been few attempts to study the design of concepts and their impact in software.

Git Overview. Version control (also referred to as revision control or source control) is any practice that tracks and provides control over changes to source code. This enables users to, for example, revert files back to a previous state, merge changes from different sources, and compare changes over time. Version control systems have become ubiquitous and indispensable tools for project coordination. Playing such a vital role in the software development process, they have the power to significantly increase the productivity of users, or become an obstacle to progress, a source of frustration, or even the cause of loss of data.

Onward! 2013, October 29–31, 2013, Indianapolis, Indiana, USA.

http://dx.doi.org/10.1145/2509578.2509584

Git is a free, open-source, distributed version control system. Designed initially for the Linux kernel development, it has since been adopted by many other open source projects including jQuery, Ruby on Rails and CakePHP [2]. Companies such as Facebook, Google and Twitter use it for their open source projects [5–7] and have adopted it internally to some extent. GitHub, a web-based service that hosts software development projects that use Git for version control has over 3 million users and over 7 million repositories.[1]

Despite its growing adoption, many users see room for improvement. In the 2012 Git User's Survey [4], the user interface and documentation were identified by respondents as two areas in need of help.[2] Of those who expressed concerns about the user interface, 71% agreed that improvements are necessary.[3] Of those who expressed concerns about documentation, 75% agreed that the available documentation needs improvement.[4] The question "What do you hate about Git?" elicited over 1500 responses, including: "too complex for many users," "requires steep learning curve for newbies," "dark corners," "the steep learning curve which makes it harder to gain more converts," "the staging area, for example, is something I rarely find useful and often find confusing."[5]

These weaknesses have spurred the development of command line wrappers and GUIs whose primary goal is to remove some of Git's complexity (most notably, EasyGit [1] and GitHub's desktop client). These make largely cosmetic changes, through more attractive user interfaces, more consistent terminology in commands and documentation, and a focus on more commonly used commands. But their success seems to have been fairly limited, and most people still use the command line interface distributed with Git as their primary frontend, or fall back to it for accomplishing more complex tasks.

We believe that the usability problems of Git run deeper, and can only be addressed by a more fundamental reworking that reconsiders the underlying concepts. This is not to say that such a reworking cannot be achieved in a lightweight fashion. Indeed, our own reworking hides Git behind a veneer just as these other tools do. The difference however, is that we are willing to completely reconsider the core concepts that are presented to the user.

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.

We present this case study in two parts: first, an analysis of Git, and second, an outline of Gitless, our redesign. Our focus, of course, is on the conceptual flaws of Git and how we are trying to remedy them in Gitless.

Gitless as an Ongoing Effort. Gitless is an experiment, not a demonstration and validation of our philosophy of conceptual design. Needless to say, Git is the result of many years of development by many people, and has been refined in response to its application by a large and expert community of users. Gitless, in contrast, is an initial research prototype that is currently only a few weeks old. Moreover, we do not claim that Gitless occupies an ideal point in the design space; indeed, in our discussion we identify several dimensions of design that might be explored, producing very different conceptual models. Of course we hope that a redesign of Git that improves its conceptual model (according to criteria that we outline in this paper) will result in a better and more usable version control system, but we regard this as a research hypothesis to be tested empirically.

Our hope is that, at the very least, our analysis will spur interest in viewing software in this way. A fuller discussion of our research program, along with a series of examples of small conceptual analyses, appears in an accompanying technical report [28].

2. Concept Design Criteria

By "concepts" we mean the constructs and notions either that the system deals with (on account of them preexisting in the world), such as "bank account" or "airline flight," or that are invented for the purpose of structuring the functions of the system, such as "paragraph style" in a word processor, or "element group" in a drawing application. These concepts, being essential to the design, are both apparent in the interface of the system with the outside world, and (usually) in the system's implementation. In contrast, we do not mean by "concept" anything that exists only at one level, whether it be in the interface (e.g., "scroll bar") or in the implement (e.g., "callback"). In our view, concepts have psychological content; they correspond to how users think about the application.

[1] As of August 2013. Data posted on `http://github.com/about/press`.

[2] See [4], Question 23.

[3] 25% a little improvement is necessary, 27% some, 19% much. 4116 responses.

[4] 35% a little improvement is necessary, 29% some, 11% much. 4080 responses.

[5] See [4], Question 24.

Conceptual design for us, therefore, is about the design of user-visible behavior, and not the design of internal software structure, and we use the term "conceptual model" for a specification that focuses on concepts rather than the details of behavior.

Brooks lists three principles as representing the notion of conceptual integrity [12]: *orthogonality* – that individual functions 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. Despite using the terms "conceptual design" and "conceptual integrity," Brooks formulates these principles in terms of *functions*. In this case study, we will apply his principles, but attempt to focus them more closely on concepts.

To illustrate this, we present an example of a violation of each of Brooks's principles, and how it might be viewed in terms of concepts.

Orthogonality. The Epson R2400 printer allows borderless printing only for some of the paper feed options. Unfortunately, the driver for the printer (presumably in an attempt to enforce this restriction) associates borderless printing with the setting of preset margins to zero, even if a large margin is used at printing time. Consequently, page presets are (invisibly) linked to paper feed options, and if you select a preset with a zero margin, the driver prevents you from selecting the rear feeder (since it is not designed for borderless printing). The choice of paper feed in the printing dialog and the choice of preset in the page setup menu are thus coupled, causing much consternation amongst users who fail even to notice whether a preset has zero margins.[6]

The concepts here are those of a borderless print, and of preset margins; the two are coupled, so that a print with a preset margin of zero is defined to be borderless, irrespective of its actual margins. A better design might have two distinct concepts, the preset margin (to which the margin chosen at print time is added), and the gutter (say), which defines the maximum printable area.

Propriety. In the Tumblr blogging platform, if you end a post (or its title) with a question mark, a dialog box appears asking whether you would like to accept "answers." It turns out that there is a concept of a blog post that is a question (and thus solicits answers). Yet such a concept is confusing and unnecessary; there is already a concept of comments for responses to posts. And in fact, the question concept seems to have confused the developers also, since the feature is buggy (and in particular, having marked a post as accepting answers, you cannot undo it).

Generality. Some cameras let the user bind particular parameter-setting actions to physical keys – in the Fuji X100s, for example, a button marked "Fn" (for "function"). When parameters are set manually, not using such a key, the user navigates through various levels of menus, first selecting a particular feature (such as ISO speed) and then subsequently navigating more deeply or actually setting values (such as setting the maximum ISO speed in auto-ISO to 1600). Unfortunately, in the x100, the function button can only be bound to the top level selection, so that pressing it does not set the parameter, but instead opens the submenu (requiring a further step to actually change the parameter). The problem here is that the concept of action is insufficiently general; both the selection of the feature at the top level and the actual setting of parameters should be treated uniformly as actions, so the user could bind the function key at any level.

We considered including a fourth criterion, namely *consistency*, which would require that actions behave in a similar way irrespective of the arguments they are presented with, or the states in which they are invoked. The Unix shell offers many examples of violations of this criterion, amusingly documented in *The UNIX Hater's Handbook* [23]. Git is rife with surprising inconsistencies; as one of the worst examples, the `commit` command behaves completely differently if files are listed as explicit targets on the command line (breaking the common Unix idiom that not specifying files is equivalent to specifying the current directory). Even though inconsistency is one of Git's flaws, we decided not to focus on it because it seems to be primarily a user interface issue rather than a deeper conceptual issue.

3. An Overview of Git

Explaining all of Git is a daunting task, and far exceeds the scope of this paper. We therefore restrict our analysis to local operations, and even there consider only a limited subset of features. Nevertheless, the subset of Git we analyze in this paper is (we believe) rich enough to serve as the cornerstone of a case study, and in fact a newcomer to Git may be surprised at the sheer amount of complexity present in such limited functionality. We assume some familiarity with Git; for a more thorough explanation, see [16, 35].

Unlike traditional version control systems (such as CVS and Subversion), Git is distributed. Rather than having a single, centralized repository, each user has a separate repository to which work can be committed even when the user is offline. 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.

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 (using the `add` command), and then to commit

[6] A workaround is to use a tiny margin, e.g., a thousandth of an inch, whenever a zero margin is required.

them to the repository (with the `commit` command). Explanations of Git use the term "tracked files" to refer to files that have been added. 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 will save the older version of the file (representing its state the last time the `add` command was called), and not the latest version.

The situation is made more complicated by the fact that tracked files may not have corresponding versions in the staging area. Following a commit, a file that had been previously added remains tracked, but the version in the staging area is removed. The term "staged file" often used interchangeably with "tracked file" is thus subtly different: in this case, we have a file that is tracked but no longer staged.

Files that are not tracked are not included on a commit. Separately, a file may be marked as "assumed unchanged." Such a file behaves for the most part like an untracked file, but will not even be recognized by the `add` command; to make it tracked again this marking has to be removed. 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.

At any one time, the user is working in a particular branch of development. 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). 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. A Conceptual Model of Git

The view of Git embodied by our discussion and model has been obtained from popular references and discussions, and from observation (especially of the output from the so-called "porcelain"[7] commands such as `git status`).

Recall that a "conceptual model" to us is a specification that focuses on concepts, not on implementation details. And that, in our view, concepts correspond to how users *think* about the application.

Due to the complexity of Git and the lack of a succinct and clear user manual (and the fact that we ourselves are not Git devotees), there are doubtless some errors in our model.

But the conceptual model conveyed by an application – in its documentation, marketing materials, implied by its user interface, and even in the culture that surrounds it – has to be regarded as inseparable from the application itself. So to the extent that consensus is missing on an application's conceptual model, it is arguably the application itself that is at fault.

4.1 An Overview of Git's Conceptual Model

Ideally, a description of a conceptual design should be implementation-independent; it should be easy to understand; it should be precise enough to support objective analysis; and it should be lightweight, presenting little inertia to the exploration of different points in the design space.

For these reasons, we have chosen (initially at least) to use a very standard state-machine model of computation, in which named actions (performed by the user or sometimes by the application) produce transitions between abstract states. The abstract state space is described by a relational data model, using the variant of extended entity-relationship diagrams developed for the Alloy modeling language [29]. The actions are crudely specified by naming them and describing their effects on the state informally. This form of description is pretty conventional; instead, we might have chosen any of the well-known "model based" specification languages (such as Z [40], B [9], VDM [32], or Alloy). Our own preference is for a diagrammatic representation of the state space, but it may not be essential. We might have used state machine diagrams (such as Statecharts [25]) instead, but for applications like Git, such a notation would not have been suitable, because it does not support richly structured state.

Concepts correspond to state components. To connect the abstract state components with the user's understanding of them as concepts we use Michael Jackson's notion of a "designation" [31]: a necessarily informal statement that acts as a kind of recognition rule. For example, in a conceptual model of an application for managing university course registrations, we would likely need a designation for the concept of "student." Designations are invariably more challenging (and more interesting) than they first appear to be; the students registered for a course, for example, might include not only regular enrolled students but also special students, visitors, and even staff and faculty members.

The concepts that form the abstract state of Git are shown in the relational data model of Fig. 1. Each box represents a set of objects, and the arcs represent relations. A large, open-headed arrow denotes a classification relationship; thus `Tracked File`, `Untracked File`, `Assumed Unchanged File`, `Ignored File` and `Ignore`

[7] As Chacon in [16], Chapter 9.1, puts it "Git was initially a toolkit for a VCS [version control system] rather than a full user-friendly VCS, it has a bunch of verbs [commands] that do low-level work and were designed to be chained together UNIX style or called from scripts." These are the commands that are generally referred to as "plumbing" commands to differentiate them with the current, more user-friendly commands referred to as the "porcelain" commands.

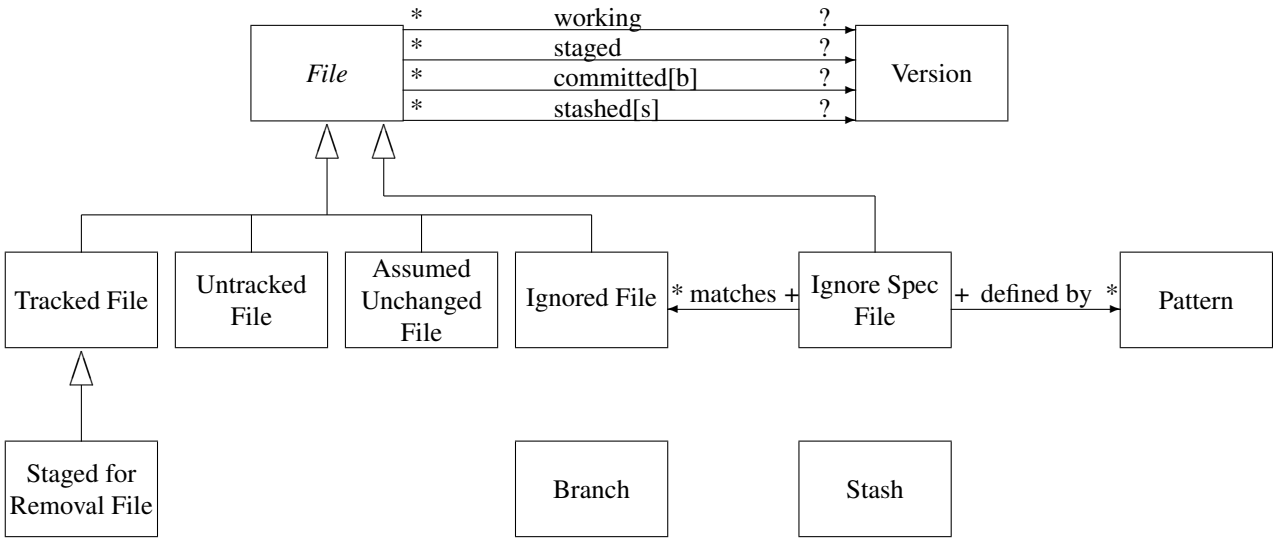

Figure 1. Graphical representation of Git's conceptual model.

`Spec File` are subsets of `File`. Moreover, the fact that `Tracked File`, `Untracked File`, `Assumed Unchanged File` and `Ignored File` share an arrowhead indicates that these are disjoint subsets of the set `File`. `Ignore Spec File` has a separate arrow to indicate that the set it represents is not necessarily disjoint from the others; a file may be, for example, both an `Ignore Spec File` and a `Tracked File`. The italicization of `File` indicates that its subsets exhaust it; that is, every file is tracked, untracked, assumed unchanged or ignored. A small, closed arrow denotes an association relationship (mathematically, a binary relation or set of ordered pairs). Thus the arc labeled `working` from `File` to `Version` associates a file with its "working" version.

The notation uses look-across cardinalities, where ! stands for exactly one, ? denotes zero or one, $*$ zero or more and $+$ one or more. So a file has at most one staged version (given by the ? at the `Version` end of the arc labeled `staged`), and each version is the staged version of zero or more files (given by the $*$ at the `File` end of the arc labeled `staged`). An arc labeled `r[i]` represents not just one relation, but a collection of relations indexed by `i`. Thus the arc labeled `committed[b]` represents an association between files and their committed version for each branch `b`. (Alternatively, as in Alloy, such a relation can be viewed as ternary; in this view, `committed` is a relation on branches, files and versions.) The cardinalities apply to each of the indexed relations; thus the markings on `committed[b]` indicate that, for each branch, a file has at most one committed version.

The diagram omits an important state component. In addition to these sets and relations, there is a scalar value – the current branch – that is used as a default for commands that can be applied to multiple branches. It should also be noted that diagrams such as this obviously do not express the full state invariants – for example, that a version must be associated by one of the relations with a file. This particular invariant is akin to the notion of a weak entity in the Entity Relationship Model [17]. 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).

4.2 Designations

We attempt to provide a clear designation for each of the concepts that form our model of Git, realizing that this will often not agree with popular accounts of Git (which are anyway often inaccurate or inconsistent).

File. A *File* is the identity of a Unix file, to be distinguished from its contents. In Git, identity is equated with the path of the file; thus the concept might equally well have been named *File Path*.

Discussion. Only files that appear within Git projects are considered, and our model takes the perspective (implicitly) of a single user's repository. When our model is extended to include synchronization actions between repositories, a single file may appear in several repositories, with different status (untracked, tracked, etc.) in each.

An alternative designation of *File* would treat it as an identity independent of the file's path, so that a change of name could be regarded as a modification of one of the properties of a file. Git sometimes provides an illusion that this is the case; by examining the contents of files, for example, the `git status` command is able to report when the name of a file has been changed (but will also incorrectly report a change of name if you copy a file and delete the original).

It is important to note that existence of a *File* does not imply that a Unix file corresponding to it exists in the file system, since a *File* can be staged for removal on the next commit.

Tracked File. A *Tracked File* is a path corresponding to a file whose modifications will be detected by Git (and reported in the `status` command).

Discussion. The Pro Git book [16] describes tracked files as "files that were in the last snapshot; they can be unmodified, modified or staged."[8] It's not entirely clear what is meant by "files that were in the last snapshot" but from the context it appears to mean "files that were in the last commit point."[9] 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 includes files that have been staged but are not part of the last snapshot, contradicting the earlier definition.

Untracked File. An *Untracked File* is a file path representing a file that appears in the working directory (see *working* below), but which has no committed version (see *committed* below). The `status` command includes untracked files in the "Untracked files" section of the output.

Discussion. This seems to correspond to the view in [16], but it is nevertheless not entirely consistent with Git's behavior. Executing `git rm --cached` of a tracked file stages the removal of the file without touching the file in the working directory, so that it will be deleted from the repository at the next commit. Subsequently running `git status` will show the file under both the "Changes to be committed" section (as "deleted") and in the "Untracked files" section. We thus have a file that is both tracked and untracked (according to our designations). Since this is an edge case, we decided to overlook it, to avoid making the model even harder to understand.

Assumed Unchanged File. An *Assumed Unchanged File* is a file path that was previously tracked, but for which the user has indicated that Git should no longer detect changes.

Discussion. We were unable to find even an attempt at designating this term in popular references. The description of the command used for marking a file as assumed unchanged says "When the 'assume unchanged' bit is on, Git stops checking the working tree files for possible modifications, so you need to manually unset the bit to tell Git when you change the working tree file."

Unlike an untracked file, therefore, an assumed unchanged file won't be listed in the "Untracked files" section of the output of the `status` command. Moreover, while adding an untracked file will cause it to become tracked, adding a file marked as assumed unchanged has no effect.

Ignore Spec File. An *Ignore Spec File* is the path corresponding to a special kind of file (`.gitignore` files) whose content is a set of patterns specifying which file paths are to be ignored. Since these are standard files, they can be tracked, untracked, assumed unchanged (and even ignored).

Discussion. A more complete model would take into account other ways Git enables a user to mark files as ignored (such as adding patterns to `.git/info/exclude` or setting up the `core.excludesfile` property for defining global ignores).

Ignored File. An *Ignored File* is a file path that is completely ignored by Git, and it will not appear in any section of the output of the `status` command. Since there can be several ignore specs in a repository (i.e., multiple `.gitignore` files in different directories) a file will be ignored if it is matched by the `.gitignore` in its directory or by any `.gitignore` file of its parent directories.

Discussion. There seems to be general consensus on what an ignored file is.

Staged for Removal File. A *Staged for Removal File* is a file path representing a file that no longer exists in the working directory and which will be removed from the repository on the next commit.

Discussion. Some subtleties of this concept are discussed below with the discussion of relation designations.

Pattern. A *Pattern* is a string specifying a set of file paths, used in an ignore spec file to define the set of ignored files, such as `*.pyc`, `*.pyo` for Python projects, `*~` to ignore Vi backup files, and so on.

Discussion. A more complete model would take into account the fact that a pattern could be of two kinds: a regular pattern or negated pattern. Any matching file excluded by a previous pattern will become included again if it matches the negated pattern.

Version. A *Version* represents the contents of a file at some point in time. This means that two distinct files with the same content would be modeled as having the same version.

Discussion. Do not confuse version with "version number," which would be a name for a version (and which does not exist as a concept in Git). Also, this concept of version does not distinguish the same content at two different places. For example, if a file was committed and the user hasn't modified it since, then in our model the working version of the file would be the same as the committed version.

Branch. A *Branch* is a named collection of committed versions of files. More precisely, in our model a branch is an identifier that indexes the relation *committed* (designated below) mapping files to committed versions.

[8] See Chapter 2.2, "Recording Changes to the Repository."

[9] In explaining what it means for a file to be untracked, the books says: "Untracked basically means that Git sees a file you didn't have in the previous snapshot (commit)."

Discussion. A more complete model would represent the structure of branching; in the simplified view we take here, there is just a collection of branches available, each with its own most recently committed versions of files.

Stash. A *Stash* is a collection of file versions that are saved together. More precisely, in our model a stash is an identifier that indexes the relation *stashed* (designated below) mapping files to stashed versions.

Discussion. Branches and stashes are conceptually unrelated, even though a stash is usually created when a user switches branches. Also, while a user can choose the name of a branch and change it at will, the stash name is assigned by Git. Stash names have the form "stash{i}" where i is the position of the stash in the stash stack from the top (a collection of all stashes). So as the stash stack grows or shrinks, the name of a given stash changes. A more complete model might include these naming issues.

Working. The *working* version of a file is the one stored in what is usually referred to as the working directory. Following a modification of a file in the working directory, *working* corresponds to the file's new content. Note that, as indicated in the diagram by the multiplicity marking, a file can have at most one working version; a file may exist but have no working version because, for example, it is committed in the repository but no longer in the working directory.

Staged. The *staged* version of a file is the one stored in what is usually referred to as the staging area, and is the version that is saved in the repository when a `commit` is performed. Note that, as indicated in the diagram by the multiplicity marking, a file can have at most one staged version.

Committed. The *committed* version of a file is the one stored in the local repository at the last commit point. Since a different version can be stored in each branch of the repository, there is a separate relation `committed[b]` for each branch; thus for a file (path) `f`, the version `f.committed[b]` is (if it exists) the version of the file `f` committed in branch `b`. Note that, as indicated in the diagram by the multiplicity marking, a file can have at most one committed version for each branch. (Of course in practice there is a separate committed version also for each commit point, but to simply our model we have chosen to ignore this.)

Stashed. The *stashed* version of a file is the one saved in what is usually referred to as the stash list. Since a different version can be stashed in each of multiple "stashes," there is a separate relation `stashed[s]` for each stash; thus for a file (path) `f`, the version `f.stashed[s]` is (if it exists) the version of the file `f` stashed in stash `s`. Note that, as indicated in the diagram by the multiplicity marking, a file can have at most one stashed version for each stash.

Discussion. Tracked files may or may not have staged versions. An untracked file, however, cannot have a staged version. When a tracked file with a staged version is committed, immediately following the commit (as explained below), the staged version is dropped but the file remains tracked. (Note that since a `git add` can be performed before a file is marked as assumed unchanged, an assumed unchanged file can also have a staged version.)

When a tracked file is removed from the working directory (using the Unix `rm` command), it will not be unstaged, and a subsequent commit will commit the last version to have been staged by the previous `git add` command. So a file can have a staged version but no working version. However, if the file is removed by the `git rm` command (discussed below), it will be staged for removal.

Some care is required in distinguishing between a file being unstaged (meaning that its staged version is removed) and being staged for removal (which means that the staging area records the absence of the file rather than its presence, so that it will be removed from the local repository on the next commit). Suppose a file is added (using the `git add` command), then removed (using the Unix `rm` command) and then committed (using the `git commit` command). Prior to the commit, the file has a staged version but no working version; after the commit, it has no staged version either. To remove the file from the repository, in contrast, one executes a `git rm`, which removes the staged and working versions, and additionally marks the file for removal in the next commit. Git's documentation is not a big help here; the `man` page for `git rm`, for example, states "Remove files from the index, or from the working tree and the index." suggesting that an unstaging is performed, when in fact a staging for removal is performed instead.

4.3 Actions

- `git add`. The `git add` f command causes the working version of file f to become (additionally) the staged version. If the file is untracked, it becomes tracked. Subsequent modifications to the file will produce a divergence between the working version and staged version.

- `git commit`. Executing the `git commit` command causes all versions of files that were staged to become committed versions. It is possible to skip the staging area altogether, and use Git in a similar fashion to traditional centralized or distributed version control systems such as Mercurial [8], using the `git commit` $f_1 ... f_n$ or `git commit -a` commands that perform an implicit `git add` prior to the commit. These variations of the `commit` command make the working version (rather than the staged version) committed. Following any form of commit, committed files are no longer associated with their previously staged versions.

- `git reset`. Executing `git reset HEAD` f for a file f removes its association to the staged version, and makes

the file untracked if it was an untracked file before the `git add` that caused it to be staged, or will make the file tracked if it previously was an assumed unchanged file. (The very same command, in the form `git reset HEAD^`, can be used for undoing the last commit, but to simply our model we haven't included the notion of a commit history thus leaving out of the model this and other variants of the `git reset` action that operate on the history.)

- `git checkout`. The `git checkout` f command reverts local changes to file f by replacing the working version, either with the staged version if there is one, or with the committed version otherwise. The same command is used to switch branches: `git checkout` b switches from the current branch to the branch named b. After a successful branch switch (no conflicts occurred), the working version of files are replaced with their committed version in branch b. If a file has local modifications (or no corresponding committed version in b), the working version (and staged version if any) remain unchanged.

- `git branch`. The `git branch` b command creates a new branch b. In addition, for each file in the current branch with a committed version, a new tuple (file, version, new branch) is added to the `committed` relationship. Conversely, when a branch is deleted (for example, by using the `-d` flag) all of its associations with files are removed.

- `git stash`. The `git stash` command takes the working version of each file (with modifications) and makes it a stashed version in a new stash. (Actually, the command also makes any staged version stashed, but our simple model does not include the fact that both the working and staged versions are stashed separately.) It will also replace the working version of each affected file with its committed version. To include untracked files in the stash, the user can pass the `--include-untracked` flag. Stashed changes can be later reapplied with the `git stash apply` or `git stash pop` commands, the difference between them being the fact that `pop` will also remove the stash (and thus the association between all files and their versions stashed in that stash).

- `git update-index`. Executing the `git update-index --assume-unchanged` f command makes file f an assumed unchanged file. A subsequent `git add` f will not make it a tracked file again; the correct command is `git update-index --no-assume-unchanged` f.

- `git rm`. If the file is a tracked file, its removal is staged and the file is also removed from the working directory. The same thing happens if the file is an assumed unchanged file but after the removal it will become a tracked file again. The command fails if the file is untracked, ignored, or if it has a staged or working version that differs from its committed version.

- `git mv`. The command `git mv old new` is a shorthand for the sequence of three commands `mv old new`; `git add new`; `git rm old`, so it adds nothing new.

- `rm`. The standard Unix remove command removes a file's working version, and has no effect on the staged (or committed) versions.

- `mv`. The standard Unix move command removes the association between the old path and its working version, and associates the new path with that working version instead.

5. 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? And before we have even considered synchronizing multiple repositories?

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. In each case we start by showing a rough edge in Git – some situation that could lead to surprising effects and user dissatisfaction – and then attempt to explain how that rough edge is rooted in the violation of the criterion.

5.1 Orthogonality

What Happened to My Changes? The purported justification for the complex concept of staged versions is to enable the user to hold a separate version of a file for committing, distinct from the working version. So, for example, the user might fix a bug, then stage the resulting version of the file (using the `git add` command), and continue to edit the file, say to expand its functionality for a much later commit. Now a vanilla commit will, as desired, save only the bug fix as the committed version.

In fact, however, the staged and working versions of a file are coupled in complex ways, and Git commands that one might expect to affect only one often affect the other too. For example, if, in the situation just described, the user executes the `commit` command to commit the bug fix, but then decides to undo the commit by executing the `reset` command, then (depending on the arguments presented) the command may not only replace the staged version with the committed version, but replace the working version too, wiping out the changes that followed the bug fix.

Why This Happens. The concepts of staged and working versions are not orthogonal. Many commands that are primarily intended to modify one of the two modify the other too. Worse, whether or not these ripple effects occur depends on which arguments are presented to the commands.

5.2 Propriety

Just Let Me Commit! The most common action performed by many users is committing changes; Git was designed in particular to make commits fast for exactly this reason. Despite this, committing files is non-trivial in Git, in large part because of the intrusion of the concept of staging.

A user might think that the working version of a file could be committed without creating an intermediate staged version. The command `git commit` f, where a file is named explicitly, causes the committed version to be obtained not from the staged version but rather from the working version. But if the file f is untracked, the command will fail with the message "pathspec did not match any file(s) known to Git."

Alternatively, the user might think that using the command `git commit -a` would enable her to avoid staging, since it performs an implicit `add` prior to the actual commit. But it also won't include untracked files, and will commit *all* tracked files with modifications (and will also remove from the committed file versions all files that have been staged for removal too). To commit all the modified files but one, the user would have to list all of those files explicitly as arguments to the `commit` command.

Why This Happens. Some Git enthusiasts make arguments for the value of the staging concept, but for most users it offers inessential functionality, and its presence complicates the use of Git considerably.

5.3 Generality

I Just Want to Switch Branches! Say the user has two branches b_1 and b_2, where b_2 is several commits ahead of b_1 and there's a committed file f_2 in b_2 that is not present in b_1. Working in branch b_1, the user is unaware of that, and creates a new file f_1 with the same name as f_2. Suppose f_1 is an untracked file, present in the working directory. Usually, changes in the working directory are kept when switching branches. As stated in the help message of the `git checkout` command: "Local modifications to the files in the working tree are kept, so that they can be committed to the <branch>." But in this case, since Git detects a conflict, the user gets an error when trying to switch to b_2.

An equivalent problem arises if f_2 is present in both branches, but branch b_2 contains changes to f_2 that are not reflected in branch b_1. Then if the user edits f_2 in b_1 and tries to switch to branch b_2, a conflict is reported and the switch fails. To work around this problem, the user must either commit (potentially saving to the repository unfinished work), or use stashing. Either way, the user needs to execute an extra command to achieve the desired goal, leading to a mismatch between the user's straightforward goal and the complex sequence of commands necessary to achieve it.

Why This Happens. Branches are intended to support independent lines of development. A line of development comprises both the working versions of files and committed versions. And yet, in Git's conceptual model, only the committed versions are organized by branch; while there are potentially multiple committed versions of a file (one per branch), there can only be one working version. There is thus a lack of generality, with the branching feature essentially available only in one area and not another.

I Just Want to Stop Tracking a File! Suppose a user creates a database configuration file for an application, and commits it. At a later time, the user might want to alter that file in order to perform local testing. This new version of the file should not be committed. How can this be achieved? Of course, the user might make the set of files explicit on every single commit (leaving out the database configuration file), but this is laborious and error-prone. You might think the file could be added to the ignore specification in the `.gitignore` file, but this won't work, since a file that has already been committed cannot be ignored.

This simple task turns out to be surprisingly challenging. It involves using the rather strange command `git update-index` to mark the file as "assumed unchanged," and since such files aren't listed in the "Untracked files" section of the `status` command's output, the user needs to remember to execute a different command, `git ls-files -v`, to find out which files are marked "assumed unchanged."

Why This Happens. The concepts of "assumed unchanged" and "untracked" play fundamentally the same role: they mark files that are not to be included in commits. And yet there is no general concept that subsumes the two of them, since Git distinguishes files according to whether they have been previously committed; a file can only be untracked if it is not also committed. Whether a file is committed is not under the user's control, since a file can be committed as a result of a pull from another user's repository. In short, there is a violation here both of generality (since a more general concept would subsume both cases), and propriety (since the concept of "assumed unchanged file" could be eliminated).

6. Gitless

Gitless is free, open-source, and distributed under GPL. It covers the most common Git use cases, including some whose analysis is out of scope of this paper (such as converging divergent branches, and syncing with other repositories). The tool can be downloaded from `http://people.csail.mit.edu/sperezde/gitless`, which also provides links to a user manual and the code repository.

6.1 An Overview of Gitless's Conceptual Model

Fig. 2 shows the relevant parts of Gitless's conceptual model. The key differences between Git and Gitless are: (a) the elimination of the concept of `Assumed Unchanged File`, and the generalization of `Untracked File` to subsume it; (b) the elimination of `staged` versions; (c) the elimination of the concept of `Stash`, and `stashed` versions; (d) indexing of `working` versions by branch. Switching branches in Gitless is equivalent to always creating a stash (more precisely, of executing `git stash --all`) before switching to a different branch in Git, and then retrieving this stash when the user switches back to the original 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.

6.2 Designations

File. The concept of a *File* in Gitless is equivalent to the concept of a *File* in Git.

Tracked File. A *Tracked File* is a file whose changes Gitless will automatically detect. Tracked files are automatically considered for commit if they are modified.

Untracked File. Conversely, an *Untracked File* is a file whose changes Gitless will not automatically detect.

Ignored File, Ignore Spec File, Pattern and Version. These concepts are equivalent to the ones in Git.

Working. The *working* version of a file is the one stored in the working directory of the user. Each branch has its own set of *working* versions of files. Following a modification of a file in the working directory, *working* corresponds to the file's new content.

Committed. This concept is equivalent to the one in Git.

Branch. A *Branch* is an identifier that maps files to *working* and *committed* versions.

6.3 Actions

- `gl track`. Executing `gl track` causes the files given as input to become tracked. No change is made to working or committed versions of files.

- `gl untrack`. Executing `gl untrack` causes the files given as input to become untracked. No change is made to working or committed versions of files.

- `gl commit`. In its default form (with no arguments given), this command commits the working version of all tracked files with modifications: for every tracked file whose working version is different from its committed version it will make the working version additionally committed. 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.

- `gl checkout`. Unlike in Git (where `git checkout` behaves differently according to whether the argument is a file or branch name), the `checkout` command in Gitless only accepts files. Its effect is to make the working version of each file listed equal to its committed version; the committed version is not affected.

- `gl branch`. Executing the `gl branch` b command sets the current branch of the repository to b, so that the working versions that appear in the working directory are the ones associated with b. It previously creates b if it doesn't exist, additionally adding, for each file in the current branch with a committed version, a new tuple (file, version, new branch) to the `committed` and `working` relationships.

- `rm`. The standard Unix remove command has the same effect as in Git.

- `mv`. The standard Unix move command has the same effect as in Git.

A more thorough description of these Gitless commands, including those that are out of scope for this paper can be found at the Gitless website.

6.4 Gitless versus Git

6.4.1 A More General Untracked Concept

Gitless eliminates the concept of an assumed unchanged file by making the concept of an untracked file more general. This addresses the rough edges described in Sec. 5.3 caused by the violation of the generality (and propriety) criteria.

We believe there's no reason to differentiate them. At their core, they represent the same thing: a file the user doesn't want to be considered for committing. The distinction made by Git not only adds conceptual complexity but also forces the user to learn separate commands – not only for marking/unmarking an assumed unchanged file, but also for listing files, since `git status` shows one category but not the other.

6.4.2 A More General Branch Concept

In Gitless, the concept of branch is more general, and includes working versions of files as well. In Git, branching supports two objectives, according to whether uncommitted

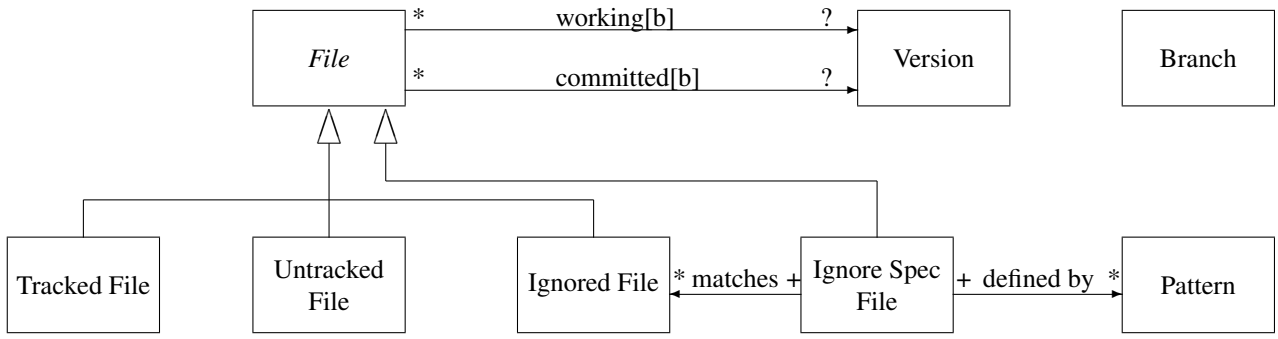

Figure 2. Graphical representation of Gitless's conceptual model.

changes are copied from the source branch to the destination branch. Gitless favors the case in which the user wants to keep the state of the destination branch independent of the state of the current branch; uncommitted versions are not copied to the new branch, but are also not lost (since they are preserved as the working versions in the old branch). We believe this to be a more common use case and that it lives up to the expectation of a branch being an *independent* line of development.

This addresses the rough edges described in Sec. 5.3 that forces the user to either commit unfinished changes or stash them before being able to switch branches.

6.4.3 No Staged Version

We have yet to be convinced that Git's concept of staging adds any benefits that can outweigh its complexity. Gitless therefore eliminates the concept of a tracked file having a staged version, and there is a single and direct path (in both directions) between working and committed versions.

The elimination of the staged relation addresses the rough edges in Git described in Sec. 5.2 caused by its violation of the propriety criteria and also addresses those described in Sec. 5.1 caused by its violation to the orthogonality criteria.

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. Gitless addresses this issue 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, we might address this by extending the `gl commit` command (for example, using a `-p` flag in `gl commit` similar to Git's `commit -p`), without requiring staging.

Admittedly, there are some use cases for which a staging area comes in handy. For example, 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 doesn't 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.

6.4.4 No Stashed Version or Stash

The main purpose of stashing in Git seems to be enabling a switch between branches when a conflict occurs. Since Gitless's concept of a branch also includes the working version of files these conflicts don't occur anymore, so the need for stashes and stashed versions is eliminated.

6.5 Evaluation

The first experimental version of Gitless was released a few weeks ago to students in our research group, who have been using it in their daily work and provided us with some initial feedback.

There was general agreement that the assumed unchanged part of Git is messy, but the improvements in that area were, for the most part, unnoticed. Untracking a file that exists in the repository seems to be a fairly unusual use case.

The elimination of the staging area was enthusiastically received as a major reduction in complexity, though one student missed being able to stage files and then only diff those staged files prior to committing (using `git diff --staged`). We believe this to be a limitation not so much in the conceptual model of Gitless but rather in the detailed functionality of the `gl diff` command, which appears to be insufficiently versatile.

6.6 Other Points in the Concept Design Space

A file in Gitless, as in Git, is essentially a file path. A richer concept would enable an identity distinct from the file's path, so that changes to a file's name or location could be treated as changes to properties of a file.

Treating files and directories as specializations of a common file system object (as in Unix) would lead to a more consistent and flexible tool. (Not surprisingly, in the 2011 Git User's Survey [3], for the question "Which of the following features would you like to see implemented in Git?"[10][11], the choice "support for tracking empty directories" got selected by 2045 respondents[12] becoming the second most desired feature of all.) But this raises tricky questions, in particular what it would mean to track a directory. Would it mean that all files under it are tracked by default? Could an untracked directory contain tracked files?

Tracking might be by name rather than file identity. In the current design, ignored files are specified by name; this makes it possible to classify a set of files as ignored before they even exist. The same strategy might be used for tracked files, with the user providing not the name of a specific file but rather a specification (e.g., as a regular expression over file paths).This might unify the concepts of untracked and ignored files. We explored this idea, but backed off when we realized that the semantics of a sequence of track and untrack commands on overlapping but not identical specifications becomes quite intricate.

An alternative to explicit tracking would be to have all files that are not matched by the corresponding ignore specs tracked automatically. All files would be considered for commit unless they are ignored. The `commit` command would need to be more flexible though, since it would have to compensate for the lack of `gl untrack`/`gl track`.

7. Related Work

Usability and Going Deeper. 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* [41]: "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* [18], in which he gives many examples of assignments he rejected because the client expected him to work only on the surface appearance.

Usability and Conceptual Models. 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 [37], 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."

Conceptual Integrity. Fred Brooks put the term "conceptual integrity" on everybody's lips when he described it in *The Mythical Man Month* [13] 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* [12] in a section entitled "Conceptual design, especially, must not be collaborative."

It's 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 [20].

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 [15], and which are recapitulated in *The Design of Design* [12]. 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 [24], 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 [12]. 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* [12], Brooks himself later identifies style, quoting Webster's

[10] Question number 17.

[11] Question wasn't asked in the 2012 survey.

[12] 33% of the total.

dictionary, as being more about "form or expression" than about "substance."

In his widely read "The Rise of the Worse is Better" [21], 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 [22], 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.

Conceptual Modeling. An entire field is devoted to "conceptual modeling" (see, for example, the textbook by Olivé [38]); 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 [17], continuing with research on more expressive "semantic models" [27], and then merging with the development of notations for more general software design, such as the object model of OMT [39], which became the class diagram of UML [11]. Arguably, the field of formal specification had the same motivation in mind – the Z language [40] 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 [29] placed more emphasis on the description of the data structure than its predecessors, and was designed to make it easier to express conceptual data models textually (principally by providing a very expressive declaration syntax to support subtyping). Also, Alloy was designed in the context of an advocacy of "lightweight formal methods" [30], which emphasized capturing the essence of a system over detailing all of its behaviors.

Bjørner's work (see, e.g., [10]) 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* [19] and in David Hay's *Data Model Patterns: Conventions of Thought* [26].

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 commonly argued that the objects comprising the system emerged almost trivially from the problem domain. Thus Bertrand Meyer in *Object-Oriented Software Construction* [36]: "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 reality being modeled, the objects are just there for the picking!" Similarly, in John Guttag and Barbara Liskov's *Abstraction and Specification in Program Development* [33], 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 [34], incidentally, introduced conceptual models for requirements.)

8. Future Work

In the near future, we plan to extend our analysis to other features of Git that were left out of scope, such as the different ways of converging changes (merging, rebasing, cherry-picking), submodules and synchronizing with other repositories. We also plan to extend our study to other version control systems. We hope to build a small but diverse community of Gitless users that will serve as experimental ground for our redesign efforts.

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, and extending and refining the criteria outlined in this paper, and applying them in more diverse case studies. We intend to build a catalog of conceptual idioms that appear repeatedly in different settings, and to note cases in which designs stray needlessly from conventional idioms. If we're lucky, we may even make some progress in the challenge that seems to have eluded the software engineering community for so many years: figuring out what conceptual integrity means, and finding a way to achieve it.

Acknowledgments

Thank you to our anonymous reviewers and to our shepherd, Dick Gabriel, whose insightful critique and guidance greatly improved this paper. Thanks also to Eunsuk Kang, Aleksandar Milicevic and Joe Near for being our first Gitless guinea pigs. This research is part of a collaboration between MIT and SUTD (the Singapore University of Technology and Design), and is funded by a grant from SUTD's International Design Center.

References

[1] EasyGit. URL `http://people.gnome.org/~newren/eg`.

[2] Projects That Use Git for Their Source Code Management. URL `http://git.wiki.kernel.org/index.php/GitProjects`.

[3] Git User's Survey 2011. URL http://git.wiki.kernel.org/index.php/GitSurvey2011.

[4] Git User's Survey 2012. URL http://git.wiki.kernel.org/index.php/GitSurvey2012.

[5] Facebook's GitHub Page. URL http://github.com/facebook.

[6] Google's GitHub Page. URL http://github.com/google.

[7] Twitter's GitHub Page. URL http://github.com/twitter.

[8] Mercurial. URL http://mercurial.selenic.com.

[9] J.-R. Abrial. *The B-book: Assigning Programs to Meanings*. Cambridge University Press, New York, NY, USA, 1996. ISBN 0-521-49619-5.

[10] D. Bjørner. Domain engineering: A software engineering discipline in need of research. In *SOFSEM 2000: Theory and Practice of Informatics*, volume 1963 of *Lecture Notes in Computer Science*, pages 1–17. Springer Berlin Heidelberg, 2000. ISBN 978-3-540-41348-6. doi: 10.1007/3-540-44411-4_1. URL http://dx.doi.org/10.1007/3-540-44411-4_1.

[11] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1999. ISBN 0-201-57168-4.

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

[13] F. P. Brooks. *The Mythical Man-Month: Essays on Software Engineering*. Addison-Wesley Reading, 1975. ISBN 978-0-201-00650-6.

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

[15] F. P. Brooks and G. A. Blaauw. *Computer Architecture - Concepts and Evolution*. Addison-Wesley, 1997. ISBN 978-0-201-10557-5.

[16] S. Chacon. *Pro Git*. Apress, 2009. ISBN 978-1-4302-1833-3. URL http://git-scm.com/book.

[17] P. P.-S. Chen. The entity-relationship model–toward a unified view of data. *ACM Trans. Database Syst.*, 1(1):9–36, Mar. 1976. ISSN 0362-5915. doi: 10.1145/320434.320440. URL http://doi.acm.org/10.1145/320434.320440.

[18] H. Dreyfuss. *Designing for People*. The Classic of Industrial Design. Allworth Press, New York, NY, USA, 2003. ISBN 978-1-58115-312-5.

[19] M. Fowler. *Analysis Patterns: Reusable Object Models*. Addison-Wesley Professional, 1996. ISBN 978-0-201-89542-1.

[20] R. P. Gabriel. Designed as designer. In *Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, OOPSLA '08, pages 617–632, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-215-3. doi: 10.1145/1449764.1449813. URL http://doi.acm.org/10.1145/1449764.1449813.

[21] R. P. Gabriel. Rise of worse is better. URL http://www.dreamsongs.com/RiseOfWorseIsBetter.html.

[22] R. P. Gabriel. Worse is better. URL http://www.dreamsongs.com/WorseIsBetter.html.

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

[24] W. Griswold. Conceptual integrity, December 1995. URL http://cseweb.ucsd.edu/users/wgg/CSE131B/Design/node6.html.

[25] D. Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, 8(3):231–274, June 1987. ISSN 0167-6423. doi: 10.1016/0167-6423(87)90035-9. URL http://dx.doi.org/10.1016/0167-6423(87)90035-9.

[26] D. Hay. *Data Model Patterns: Conventions of Thought*. Dorset House, New York, NY, USA, 1996. ISBN 978-0-932633-74-3.

[27] R. Hull and R. King. Semantic database modeling: Survey, applications, and research issues. *ACM Comput. Surv.*, 19(3):201–260, Sept. 1987. ISSN 0360-0300. doi: 10.1145/45072.45073. URL http://doi.acm.org/10.1145/45072.45073.

[28] D. Jackson. Conceptual design of software: A research agenda. Technical Report MIT-CSAIL-TR-2013-020, MIT, 2013. URL http://hdl.handle.net/1721.1/79826.

[29] D. Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, Cambridge, MA, USA, 2006. ISBN 0-262-10114-9.

[30] D. Jackson and J. Wing. Lightweight formal methods. *IEEE Computer*, 29(4):21–22, April 1996.

[31] M. Jackson. *Software Requirements & Specifications: A Lexicon of Practice, Principles and Prejudices*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1995. ISBN 0-201-87712-0.

[32] C. B. Jones. *Systematic Software Development Using VDM*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, second edition, 1990. ISBN 0-13-880733-7.

[33] B. Liskov and J. Guttag. *Abstraction and Specification in Program Development*. The MIT Press, Cambridge, MA, USA, 1986. ISBN 0-262-12112-3.

[34] B. Liskov and J. Guttag. *Program Development in Java: Abstraction, Specification, and Object-Oriented Design*. Addison-Wesley Professional, 2000. ISBN 978-0-201-65768-5.

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

[36] B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, second edition, 1997. ISBN 0-13-629155-4.

[37] D. Norman. *The Design of Everyday Things*. Basic Books, New York, NY, USA, 2002. ISBN 978-0-465-06710-7.

[38] A. Olivé. *Conceptual Modeling of Information Systems*. Springer, 2007. ISBN 978-3-540-39390-0.

[39] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1991. ISBN 0-13-629841-9.

[40] J. M. Spivey. *The Z notation: A Reference Manual*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989. ISBN 0-13-983768-X.

[41] B. Tognazzini. First principles of interaction design. URL `http://www.asktog.com/basics/firstPrinciples.html`.