FOREWORD

LET ME TELL IT to you straight: Git is infuriating.

Wait! Don't run off just yet. Because while Git *is* infuriating, it's also critical in two very different and equally compelling ways: first, speaking practically, Git is a prerequisite for collaborating on websites or applications, which, if you're holding this book, is probably something you are wont to do. And second, Git is a kind of model for present-day collaboration—that is, collaboration among distributed teams, working asynchronously, on a shared body of work.

So while you don't have to love Git, you do have to *know* it.

Many Git tutorials bend over backwards to map Git's arcane practices on to real-world phenomena, often leaving readers hanging from trees wondering which branch is about to snap. Here, David Demaree dispenses with that nonsense, inviting you to learn about Git on its own terms. Rather than extended, creaking metaphors, Demaree patiently explains in plain language the core principles underlying Git that every designer, developer, content strategist, and product manager needs to know. The result is a brisk, clear book you can read in a few hours and then return to your terminal, ready to confidently pull and merge.

But there's more here, and you'd be wise not to miss it. Along with the commands and syntax, there's keen advice in these pages about working on a team. Knowing when and how to commit a change is more than just a means of updating code—it's also a practice for communicating and sharing work. It's a process, and a remarkably powerful one. So while Git's quirks often leave newcomers reaching for drink, its influence on people who make websites is well deserved. By all means, devour the following chapters in order to understand how to manage merge conflicts and interpret a log. But don't forget that Git's ultimate audience isn't machines—it's humans.

—Mandy Brown

INTRODUCTION

WHEN I STARTED making websites as a hobby in 1995, being a web developer meant knowing HTML. That's it. Neither JavaScript nor CSS would ship in browsers for a year, and Flash wouldn't exist until later in the decade. The web was just starting to become a rich medium full of engrossing content, and anyone with a text editor who could remember a dozen or so tags could participate. It was nice.

Twenty years later, web development is no longer so simple. HTML, CSS, and JavaScript remain the foundation of our work, but over their history—their recent history in particular—they've evolved from languages for crafting documents, simple enough that most designers could write them from memory, into a platform for writing applications. It feels like we don't make web *pages* anymore; we make *themes* or *templates* or, if we're really ambitious, we make *apps*. We're producing thousands of lines of increasingly complex code, and we're sharing responsibility for managing that code with more people, in more and more ways. We have the power to make truly amazing things for our users, things we never could have imagined when the web was young—but at the cost of feeling like gerbils running on a technology treadmill.

Frank Chimero put it well (http://bkaprt.com/gfh/00-01/):

> *Now is the time to come clean: GitHub is confusing, Git is confusinger, pretty much everything in a modern web stack no longer makes sense to me, and no one explains it well, because they assume I know some fundamental piece of information that everyone takes for granted and no one documented, almost as if it were a secret that spread around to most everyone sometime in 2012, yet I somehow missed, because—you know—life was happening to me, so I've given up on trying to understand, even the parts where I try to comprehend what everyone else is working on that warrants that kind of complexity, and now I fear that this makes me irrelevant, so I nestle close to my story that my value is my "ideas" and capability to "make sense of things," even though I can't make sense of any of the*

above—but really, maybe I'm doing okay, since it's all too much to know. Let the kids have it.

Git is hardly the most complicated new web technology, but it's a part of this new stack that all parts of the stack have in common. You cannot escape Git if you want to participate in the new platform-y web. At some point you'll need to contend with it, either directly or as a transport mechanism used by some other tool. And that may very well be why Git is a poster child for this sea change in how we make websites.

Plenty of books, blog posts, and other online materials have cropped up to teach users at all levels how to use Git. Yet despite this wealth of tutorials, some days it feels like you can't turn around without bumping into someone complaining that Git makes no goddamned sense. And yet we use it. It seems like we have to use it, despite fearing that we cannot confidently use it, leaving us to feel like we're running around the house with a big pair of scissors.

And it's not just designers like Frank Chimero. Folks who are new to the web, or who want to work in fields only tangentially related to web development (like writing or open data), are also forced to live on a Git planet, as are tons of us who like the engineered web just fine, but who still feel flummoxed by Git.

Having spent most of the last decade using Git on almost every project, delving at times into some of the darkest, weirdest corners of Git behavior, I can safely say that *it's not you, it's Git*. Git isn't difficult because you're not smart enough, or because you missed an important meeting. Git is difficult because *Git is difficult*.

Git is difficult, in part, because it embodies what Joel Spolsky calls a "leaky abstraction" (http://bkaprt.com/gfh/00-02/). Abstractions, in the software sense, are things that make a task conceptually easier to handle by covering up the elements that make them hard. Interfaces are abstractions: there's absolutely no relationship between dragging a file to a trash can icon to delete it and actually removing the file from your hard drive, except that a designer thought it would make the concept of deleting files easier to understand. And it works! Even though I used computers for more than a decade without knowing

how the shift key worked (seriously!), I never had trouble understanding how to get rid of a file.

Abstractions are there to protect us from complexity. A *leaky* abstraction fails at its job by letting some of the underlying complexity peek through, the same way a leaky umbrella fails at its job of keeping you dry. To quote one of Spolsky's examples:

> *You can't drive as fast when it's raining, even though your car has windshield wipers and headlights and a roof and a heater, all of which protect you from caring about the fact that it's raining (they abstract away the weather), but lo, you have to worry about hydroplaning... and sometimes the rain is so strong you can't see very far ahead so you go slower in the rain...*

Git's interface is "leaky" because its command-line interface fails to protect you, the user, from knowing how it works under the hood. And one reason why Git is so scary, is that it has its own internal logic, which doesn't always map to how we humans are used to organizing information. So knowing how it works is sometimes essential to getting it to work. To use Git successfully, you sometimes need to be able to apply Git logic to situations where human logic (and Git's supposedly human-friendly abstractions) fails. In other words, to master Git, you have to think like Git.

I want to help you understand how Git thinks.

Believe it or not, Git's challenging conceptual model is a feature, not a bug. Using Git feels like running with scissors because it's a powerful tool that will let you bend time and space to your will, which sounds like—and is—a lot of responsibility to put in the hands of mere humans. But Git believes in your ability to handle such might, and so do I.

Let's get started.

1 THINKING IN VERSIONS

IF YOU'VE BEEN AROUND FOR A WHILE, you may recall that it was once common for authors to carve their words into stone. Leaving aside the stamina this required, the size, weight, and expense of the material made it somewhat inconvenient for writers to make changes to their work once it was completed. Fixing a typo, let alone clarifying one's message or making the language flow better, required cutting away sections of stone or finding a different stretch of cave wall to write on. Even on the rare occasion when it was truly necessary to change something, it was hard—physically hard—to hold on to old revisions, making it almost impossible to compare the finished version of a poem, recipe, or cave painting with the version that preceded it, or to experiment with alternate drafts. Writing anything down at all seemed like magic.

Over the centuries, it became easier to put things in writing, which in turn made it much easier for writers to explore different approaches or to change their minds, both during the creative process and after the fact. This had the added perk of making ideas and language easier to disseminate, which made just about everyone (at least potentially) a writer.

But until the introduction of computers, the best way to record or distribute an idea was still to inscribe it on a physical object, like a piece of paper, which took time and cost money. The expense of making additional versions made it so that creating a first draft of anything—a novel, a blueprint, a painting, even a photograph—had an air of finality about it.

Many of us still think this way about how we produce our work. Taking the time to clarify and improve something over the course of multiple drafts feels like a luxury. Computers and networks have made it infinitely cheaper to spread information, but iteration still requires two things: time and discipline.

When I was in school, a few teachers attempted to show me and my fellow students the value of iteration (not to mention starting projects earlier than the night before they were due) by asking us to turn in not only our final papers, but also the drafts that preceded them. Rather than turn us into nimble iterative thinkers, though, mostly this kept us up late scrambling to meet deadlines a few times a semester instead of just once. Given the choice between spending extra time making three versions of an essay versus one—even though in doing so we'd make each one better than the last—we'd much rather be lazy, settle for flawed or mediocre work, and spend our time catching up on old episodes of *Fringe*.

But there are at least two areas of our written culture where making incremental changes, and tracking those changes across multiple versions, is not just helpful but crucial: law and (more important for our story) software source code.

Like other kinds of writing, source code went through what might be called an analog phase. Early computers had to be programmed by punching holes into cards, which were fed individually into the machines, which in turn performed the instructions encoded into the cards and returned a result. (The computing words "bugs" and "debugging" are popularly attributed to Grace Hopper, who traced problems in the operation of the Harvard Mark II computer to moths that had nested among the data relays.)

Early coders endured some of the same problems that early writers did: making changes on physical media like punch cards was time-consuming and expensive. Their programs took hours

or days to run, and an error in a program meant that the whole sequence needed to be restarted from scratch, making it very important to get it right the first time whenever possible.

Computing languages have to be understood by machines, which—science fiction notwithstanding—remain much stupider than we are. Where a human can read "their" instead of "they're" (or the code equivalent) and just sigh at an author's poor attention to detail, a computer will crash. A computer system that crashes is not very useful, so software makers tried to make things easier the best way they knew how: by building more systems.

ELEMENTS OF VERSION CONTROL

What they came up with is a *version control system*. The basic principle of version control is this: instead of keeping only the latest copy of something, you hold on to each successive revision as you work, so that you can refer or revert back to an older version if you need to. Although you can use software tools—one of which is the subject of this book—to help you, version control is more importantly a *practice*. It's something we *do*—not just the tools we use to do it.

Many of us have had projects where we kept copies of old versions of our work, saving new versions by using an app's Save As... command to give each new copy its own name. Perhaps we marked the new filename with the current date (project_2014-04-15.doc), or maybe we added a version number (mockup-1a.psd). Both are rudimentary, but entirely valid, forms of version control.

Version control systems like Git work by keeping a copy of each successive version of your project in something called a *repository*, into which you *commit* versions of your work that represent logical pauses, like save points in a video game. Every commit includes helpful metadata like the name and email address of the person who made it, so you can pinpoint whom to praise (or blame) for a particular change. These commits are organized into *branches*, each representing an evolutionary

track in your project's history, with one branch—the trunk, or master branch—representing the official, primary version. Having built up a history of past commits, it's easy to retrieve any previously committed version of your project, roll back changes, or compare two or more versions to aid in debugging.

In order to save changes to your repository, there needs to be one version of the project that you can safely make changes to. Version control systems like Git usually call this the *working copy*. Its job is to act as a scratch pad for any changes you may want to make to the project; you'll eventually commit these changes to the repository as an official, saved version. From our perspective, a working copy is usually easy to spot—it's the copy of the project that appears on our hard drives, as regular files.

Version control can seem laborious, because in a regular desktop workflow, we're expected to save changes to our work twice: once to the working copy, then *also* to the repository. As a young web developer starting out, I found this annoying enough to avoid version control altogether. Eventually, though, I came to appreciate the benefits of having every significant version of my projects stored, annotated, and neatly organized in a secure location. It also helped me to think of commits as *significant* changes, as opposed to the hundreds of little changes I might save in a given hour. The extra steps involved in committing—the brief pause from coding, having to write a descriptive message, occasionally having to stop and address conflicts between my version and someone else's—have ultimately helped me develop a more thoughtful and judicious way of working.

Although adopting a basic version control practice is a little extra work, it's not *hard* work. But like anything we do to stay organized, version control works best when it's practiced consistently, so it can become what *Getting Things Done* author David Allen calls a "trusted system." On one hand, once you've committed a version of your work to a repository, you should be able to trust that when you look for that version later, you'll be able to find it in exactly the same state as when you committed it. (As we'll see, Git has that part well covered.) But you

should also be able to trust that the version you're looking for was committed to begin with, which means committing *yourself* to committing your changes at regular intervals as you work.

COMPLEX PROJECTS

Versions of single files, like Photoshop documents, are easy to manage: each represents a complete copy of the project as it existed at a certain point in its history, and (if you're using numbers or dates to identify versions) it's easy to tell when that point was by just scanning the list of versioned file names. But while some things we work on are neatly encapsulated in single files, others—like websites and apps—consist of entire directories of source files. How can we apply version control to projects like that?

This is where software version control systems like Git really earn their place in our toolbox—in managing more complex projects like websites or the source code for an app, or when coordinating changes from lots of collaborators.

The simplest version control method is the same one we'd use for a Photoshop file: make a copy of the whole project directory, appending dates or incremental version numbers as we go along. The directory named `our-website/versions/v12` is the twelfth revision to our project. Just as with a single file, every time we make a significant change to the website, we'll create a new numbered copy of the whole project: the one after v12 is v13, followed by v14, and so on. Here, the `versions` directory acts as our repository, and each numbered directory is a committed version. In simple cases, this works just fine. Indeed, before I got into using version control systems, that's how I managed web projects for clients.

Because a website consists of many different kinds of files, we need our working copy to be saved to the hard drive as its own directory: `our-website/working-copy`. The process for committing changes to this project is a bit more complicated than for our hypothetical Photoshop file, but only a bit: we make and test changes to the website in the `working-copy` directory; then, when we're ready to publish, we commit those

changes by making a copy whose name includes the next version number in sequence, such as `versions/v13`.

But what happens when we try to share this version control system with other people?

These days, sharing files—such as our store of committed versions—is the easy part. Your repository can be a shared folder on a service like Dropbox or Google Drive, or a networked hard drive or file server in your office. Inviting a new collaborator to your project can be as simple as granting them access to that folder. Everyone who has access to the shared folder can potentially commit changes.

Where things get tricky is not simply in syncing changes between teammates, but in *coordinating* those changes so that your version control system remains trustworthy and viable. This system of numbered, versioned directories and working copies may seem straightforward, but you can never assume that two people will interpret or follow simple rules the same way. To sustain a trustworthy system, it's essential that the rules *always* be followed in *exactly* the same way. If you work on a small team and are thinking, "Come on, it's not that scary," imagine having to explain, let alone implement, a system like this within a large corporation, or an open-source project like Linux with *thousands* of contributors. This isn't just a random example: Git was invented by Linus Torvalds precisely to meet the demanding needs of the Linux project, after a licensing dispute about the commercial version control tool they were using (http://bkaprt.com/gfh/01-01/).

For the sake of argument, though, let's suppose that everyone on your team understands the rules, and that any member of the team can reliably commit a new version. Here's where things get *really* hairy: what happens when two people want to commit new versions *at the same time*?

The difficulty here doesn't involve following the rules so much as communication. Once we reach the point where two collaborators might need to work with the same files at the same time, we run the risk of one person's changes overwriting—or, to use my preferred technical term, "clobbering"—another's person's work.

Imagine you and I are collaborating on a website. You're exploring what it would look like if we changed all the links on the site from blue to green, while I am considering changing the link color to red. Under the rules of our version control system, I should make and save my changes to the copy of the stylesheet file in the directory. Unfortunately, the rules also say you should make *your* change in the exact same place. Following this method, whether the links are going to be green or red in the next numbered version of the project depends entirely on which of us made our change last. If I saved my file later than you saved your file, my changes win.

The only way to avoid this clobbering is for individuals interested in submitting changes around the same time—you and me, in this case—to work together to make sure that either our changes don't conflict or that our version control system somehow automatically reconciles or rejects conflicting changes.

Some primitive version control systems solve this problem by requiring people to "check out" a file, like a book from the library. Checking a file out marks it as uneditable by anyone else until the person who checked it out has both finished editing *and* explicitly checked it back in. This addresses the risk of unintended file-clobbering (by making accidental overwrites impossible), but it also creates a new problem: if I'm the one with the homepage file checked out, you're stuck waiting on me to finish before you can do your work.

DISTRIBUTED COLLABORATION

Instead of having one working copy shared by everyone, we can require all team members to have their own working copies stored on their own computers. In theory, at least, that allows each of us to work independently until it's time to save a new official version.

There's still a small risk of two people trying to commit versions at the same time, but that happens less frequently than just saving changes while working, and we can coordinate those kinds of changes easily—"Hey, I'm going to push version

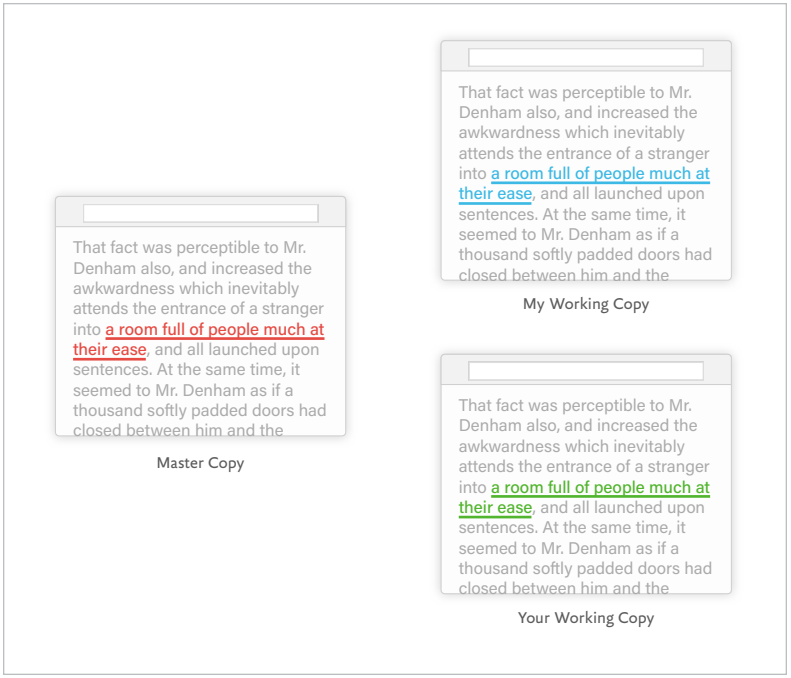

FIG 1.1: Three different copies of this web page have different link colors. How can we know which one is the correct one?

34 of the website, everyone cool with that?"—via email or a tool like Slack.

With this system, we haven't done anything to try to merge together different versions, with different changes, into a cohesive whole. Rather, we're just making named copies of folders and assuming that our working copy is a trustworthy, canonical source for the next version. The next big problem to solve is what happens when that stops being true, and our working copies drift out of sync.

In the diagram in **FIG 1.1**, your version of the site now has green hyperlinks, while my copy has red links, and the latest official version (v12) of the website has blue links. Both working copies are newer than the shared master copy, but beyond that,

how can we know what color the links should be in the next official version?

More importantly, how can we know what *else* might have changed? What if, in addition to the changed link color, your copy of the website includes a lovely new page explaining the company's mission that isn't in my copy, and mine has a fix for an annoying JavaScript bug that isn't in your copy?

This is where our homegrown version control system completely breaks down. It's not that we can't work together to combine our two working copies into a single version; it's that doing so takes up valuable time we'd much rather spend writing, designing, coding, making coffee, or browsing Tumblr for cat GIFs. Auditing our work files for conflicting changes is no fun; more importantly, it doesn't scale. As we add more files, more collaborators, and more changes, the risk of accidentally introducing major problems increases.

Remember that version control is more than just the versions: it's the rules and processes for *managing* versions. Once you have a lot of files, a lot of collaborators, or both, it can be exhausting to enforce those rules without a referee—and only by enforcing the rules consistently can you trust your version control system enough to get any value out of it.

Fortunately, all of these things—enforcing rules, keeping track of versions in a repository of past work, shuttling changes back and forth between the repository and your working copy, even merging together two directories and policing conflicts—are things that computers can do a lot faster and better than we can. By learning and adopting an *automated* version control system like Git, we can keep our work neatly organized and our changes safely coordinated with one another, all without a lot of effort—that is, once we adopt and learn how to use such a system.

HUMANS, MEET GIT

Git does for version control what web standards have done for our code or, for that matter, what Microsoft Word has done for word-processing documents. Because Git is so ubiquitous, once

you know it, you can send code almost anywhere. Git excels at synchronizing changes between different computers, whether servers like GitHub or your colleagues' laptops. Far-flung members of your team can use Git to combine efforts on a project, pushing changes to a central hub where collaborators can pull down their own copies or review work in progress, and then use Git to push changes to a web server for deployment.

Git's way of staging changes and managing branches gives you unparalleled control and flexibility over how changes to your projects are committed and organized. These attributes make Git perfect for projects like the Linux operating system kernel (http://bkaprt.com/gfh/01-02/), with thousands of contributors and *hundreds of thousands* of commits. But Git also scales down beautifully for smaller projects and teams. Whether you're looking to add version control to your personal site or share code with your whole company, learning Git gives you a seat at a very awesome table.

Note, though, that Git isn't the only tool out there for automating version control or syncing files between collaborators. People sometimes rely on simpler file-syncing services, such as Dropbox, which offer shared folders and the ability to view and restore old versions of a given file. If most of your work involves single files—Word documents, spreadsheets, PSDs—and something like a shared Dropbox folder is working for you, you may not need Git.

HOW GIT WORKS

Git keeps your project in a local repository (usually a hidden folder on your hard drive). This is an important distinction between Git and older version-control systems like Subversion or the Dropbox-based versioning scenario mentioned earlier. These server-based processes are *centralized*, in the sense that the only place you can get at your whole history of prior versions is a shared, remote space, and only your working copy is accessible offline.

Git is a *decentralized* version control system. Both your working copy and a complete copy of the entire history of the project reside on your machine, the server, and every other computer that hosts a copy of the project. By default, Git's hidden repository folders live inside a visible working copy folder. If you browse that directory, you'll see only the files and folders you expect to see in a working copy of your project. This working folder is where you'll make your changes.

Whenever you're ready, you can easily move changes into the safe, stable repository by making a commit. In our semi-manual process, we "committed" a new version by making a copy of our working copy, naming it with the next sequential version number. Committing changes in Git is, conceptually at least, very similar. For each commit, Git records the precise state of our files as they are right now in the repository for later access and retrieval. Unlike in our manual example, where we had the annoying (and potentially risky) responsibility of making sure new versions were copied into place correctly without clobbering anyone else's efforts, Git automates all of that busywork. Even better, Git copies new versions incrementally, making references to existing copies of files that haven't changed to conserve disk space.

Git not only takes care of safely copying data back and forth between the working copy and the repository (and between the local and server repositories), but it also provides a robust system for referring to different versions of the project. One of the small costs of establishing a manual version control practice is needing to decide, and then communicate to your teammates, the correct way to identify single versions. Should you use a number (like v12), or a date stamp (2014-07-28), or something else? Git allows you to assign your own names or numbers to versions if you need to, but it also gives you a reliable, unique identifier for every single commit. If you don't need to assign custom names or numbers to versions, you can just sit back and rely on Git to do that.

Finally, Git also offers powerful tools for safely merging changes between different versions of a project—not just between different collaborators, but also between multiple variations of the project on one person's computer.

THE CHALLENGE OF GIT

Version control can be challenging for newcomers not (just) because it makes things complicated, but because change is legitimately complex. Using a tool like Git forces you to question your own assumptions about how change works.

For example, one of the things version control demands of us is a nuanced understanding of *state*. As humans working in a virtual space, we're used to applying physical metaphors as handy cognitive shortcuts for understanding digital things. Let's go back to our scenario of changing the link colors on a web page. Before our minds were trained in the philosophy of versions, we thought we had a file (like it was a physical object that just happened to be on the wrong side of a computer screen), and we were *changing* it. The CSS file remained constant, but the link color changed.

In fact, from the computer's (and Git's) perspective, there are at least *three* files: the saved copy from before we made the change (with blue links), the working copy where we replaced the line that controls link color, and then (finally) the new saved copy that replaces the old one.

But nothing about the mechanics of how this one line is updated changes the fact that styles.css appears to be one file *to us*. Semantically, viewing the pile of bits named styles.css as a single thing that changes is very valuable, because it helps us understand where to find our data. Having to spend too much time concerning ourselves with the difference between the versions of this file is annoying: it's better if we can rely on the name to tell us what file this is, and have some other way of understanding how it has evolved.

It's more accurate to say that, rather than three different files, we're talking about *the same file* in three different *states*. It's the same file because even though its contents may change, its name stays the same; *logically*, therefore, it's the same file.

One potentially confusing difference between our numbered file/directory names example and a true version control system like Git is that there is no giant folder full of old versions to look at. As Git users, we're expected to know that behind each *logical*

copy of a file in our working tree, Git is safely storing all the old versions of the file, in each of its previous states.

We can comfortably understand a system where two files or directories are copies of each other, where one is a little newer or more evolved than the previous one, because there's an obvious real-world equivalent: manuscripts have second drafts, books have second printings, and so on. The rudimentary version control method I described earlier was relatively easy to understand because we were simply moving files around on a computer, something we've all done many times before. This new model feels less like writing drafts and more like time travel. It kind of *is* like time travel—and as anyone who has seen *Back to the Future Part II* can attest, time travel is complicated business.

For files saved on our hard disks, our apps and operating systems do a fantastic job of hiding—abstracting—all of that complexity from us. Instead of a flurry of versions moving back and forth between hard disk and memory, we just see an icon. Sometimes its contents change, and its "last saved" time is incremented accordingly, but visually and semantically it acts like the same file the whole time.

Not only does Git do a poor job of hiding that kind of complexity, it barely even tries. Git suffers from what I like to call an excess of simplicity.

Unlike many of the tools we use every day, Git doesn't do much to map the things it does to familiar metaphors or symbols, the way OS X maps deleting a file to the act of dragging it into the trash. Git's design assumes that you not only know how version control systems work, but specifically how *Git* works. You're meant to interact with Git on its own terms.

Git has more than seventy-five command-line functions, every single one of which has a specific job, with specific inputs and expected results. There is no single "Save new version" command in Git. Instead, to make a new commit—which is sort of, but not exactly, the same thing as saving a new version—you need to perform two or three different actions. Each of those actions has a legitimate purpose in Git-space, but none of them maps to something you'd logically do to a file or document in the real world.

But Git is also one of the most matter-of-fact programs around. It never does more than you tell it to do (though it can be easy to accidentally tell Git to do more than you wanted it to do). On one hand, this means that we might need to speak to it in more laborious, stilted terms than we're used to, which is itself an almost radical notion in an age when software can recommend a movie or summon you a taxi. On the other hand, the fact that the scope of a given command is limited means that there's also a limit to how much damage you can do at any one time. If you *do* get into Git's version of trouble—like if Git can't easily reconcile conflicting changes, or if it's uncertain about where to commit your work—there is *always* a command that will get you out of it, often with whatever work you were trying to save still intact.

As we've seen, a good argument can be made for even small teams to use version control. But the internet has made it easier than ever for people on opposite sides of the globe to work together on all kinds of projects. The open source movement has taken that even further by creating opportunities for thousands of strangers to contribute changes to projects seen and used by millions—collaboration on a wildly unprecedented scale. When you're trying to accept contributions from a community of thousands, version control becomes an absolute necessity.

So although version control may have started out as a form of insurance against mistakes, tools like Git have helped transform it into something much more compelling. As both a practice and a set of tools, version control offers us a common framework for collaborating on and sharing all kinds of work with anyone, anywhere—not to mention a new way of understanding and managing change. Thinking and working in versions not only helps us understand how projects evolve over time, but also gives us more say in how that evolution happens.

Now, how can we begin to incorporate thinking in versions into our workflow?