

# Avoiding Git Disasters: A Gory Story

Submitted by rfay on Sun, 2011-01-30 12:10

[Planet Drupal](#), [git](#)

*Edit 2015-08-30: The bottom line years later: Use the (Github's) Pull Request methodology, with a responsible person doing the pulls. You'll never have any of these problems.*

I learned the hard way recently that there are some unexpectedly horrible things that can happen to a project in the [Git source control management system](#) due to its distributed nature... that I never would have thought of.

There is one huge difference between Git and older server-based systems like Subversion and CVS. That difference is that there's no server. There's (usually) an authoritative repository, but it's really fundamentally just a peer repository that gets stuff sent to it. OK, we all knew that. But that has some implications that aren't obvious at first. In Subversion, when you make a change, you just push that change up to the server, and the server handles applying just that change to the master copy of the project. However, in Git, and especially when using the default "merge workflow" (I'll write about merge workflow versus rebase workflow in another article), there are times when a single developer may be in charge of (and able to unintentionally break) the entire codebase all at once. So here I'm going to describe two ways that I know of that this can happen.

## Disaster 1: git push --force

A normal push to the authoritative repository involves taking your new work as new commits and plopping those commits as-is on top of the branch in the repository. However, when a developer's local Git repository is not in sync with (or up-to-date with) the authoritative repository (the one we normally push to), then it can't do a fast-forward merge, and it will balk with an error message.

The right thing to do in this case is to either merge your code with a git pull or to rebase your code onto the HEAD with git pull --rebase, or to use any number of other similar techniques. The absolutely worst and wrong-est thing in the whole world is something that you can do with the default configuration: git push --force. A forced push *overwrites* the structure and sequence of commits on the authoritative repository, throwing away other people's commits. Yuck.

The default configuration in git, that git push --force is allowed. In most cases you should not ever allow that.

How do you prevent git push --force? (thanks to sdboyer!)

In the bare authoritative repository,

```
git config --system receive.denyNonFastForwards true
```

## Disaster 2: Merging Without Understanding

This one is far more insidious. You can't just turn off a switch and prevent it, and if you use the merge workflow you're highly susceptible.

So let's say that your developers can't do the git push --force or would never consider doing so. But maybe there are 10 developers working hot and heavy on a project using the merge workflow.

In the merge workflow, everybody does work in their own repository, and then when it comes time to push, they do a git pull (which by default tries to merge into their code everything that's been one on the repository) and then they do a git push to push their work back up to the repo. But in the git pull all the work that has been done is merged on the developer's machine. And the results of that merge are then pushed back up as a potentially huge new commit.

The problem can come in that merge phase, which can be a big merge, merging in lots of commits. If the developer does not push back a good merge, or alters the merge in some way, then pushes it back, then the altered world that they push back becomes everybody else's HEAD. Yuck.

Here's the actual scenario that caused an enormous amount of hair pulling.

- The team was using the merge workflow. Lots of people changing things really fast. The typical style was
  - Work on your stuff
  - Commit it locally
  - git pull and hope for no conflicts
  - git push as fast as you can before somebody else gets in there
- Many of the team members were using Tortoise Git, which works fine, but they had migrated from Tortoise SVN without understanding the underlying differences between Git and Subversion.
- Merge conflicts happened fairly often because so many people were doing so many things
- One user of Tortoise Git would do a pull, have a merge conflict, resolve the merge conflict, and then look carefully at his list of files to be committed back when he was committing the results. There were lots of files there, and he knew that the merge conflict only involved a couple of files. For his commit, he *unchecked* all the other files changes that he was not involved in, committed the results and pushed the commit.

- The result: All the commits by other people that had been done between this user's previous commit and this one were **discarded**

Oh, that is a very painful story.

How do you avoid this problem when using git?

- Train your users. And when you train them make sure they understand the fundamental differences between Git and SVN or CVS.
- Don't use the merge workflow. That doesn't solve every possible problem, but it does help because then merging is at the "merging my changes" level instead of the "merging the whole project" level. Again, I'll write another blog post about the rebase workflow.

## Alternatives to the Merge Workflow

I know of two alternatives. The first is to rebase commits (locally) so you put your commits as clean commits on top of HEAD, on top of what other people have been doing, resulting in a fast-forward merge, which doesn't have all the merging going on.

The second alternative is promoted or assumed by Github and used widely by the Linux Core project (where Git came from). In that scenario, you don't let more than one maintainer push to the important branches on the authoritative repository. Users can clone the authoritative repository, but when they have changes to be made they request that the maintainer pull their changes from the contributor's own repository. This is called a "pull request". The end result is that you have one person controlling what goes into the repository. That one person can *require* correct merging behavior from contributors, or can sort it out herself. If a contribution comes in on a pull request that isn't rebased on top of head as a single commit, the maintainer can clean it up before committing it.

## Conclusions

Avoid the merge workflow, especially if you have many committers or you have less-trained committers.

Understand how the distributed nature of git changes the game.

Turn on `system.receive.denyNonFastForwards` on your authoritative repository

Many of you have far more experience with Git than I do, so I hope you'll chime in to express your opinions about solving these problems.

Many thanks and huge kudos to [Marco Villegas](#) (marvil07), the Git wizard who studied and helped me to understand what was going on in the Tortoise Git disaster. And thanks to our Drupal community Git migration wizard [Sam Boyer](#) (sdboyer) who listened with Marco to a number of pained explanations of the whole thing and also contributed to its solution.

Oh, did I mention I'm a huge fan of Git? Distributed development and topical branches have changed how I think about development. You could say it's changed my life. I love it. We just all have to understand the differences and deal with them realistically.

## 55 comments

### Another method, fetch && merge

by [Michael Prasuhn](#) on Sun, 2011-01-30 19:04

This workflow confused my quite a bit when I was getting started with Git and one of the techniques that I found to make it more clear to me what is actually happening, is to separate git pull into git fetch and git merge.

The first benefit is that this will tell me if there are any upstream changes before starting the merge, I can check the output of the fetch to see if I'm up to date, or if there are changes to be merged. This also gives me a change to branch, and then perform the merge in a separate branch in case I want to test the separately before merging back into the primary integration branch before pushing.

This is very similar to how pull works, but for the way my mind works, it helps to know that I am independently in control of each step of the process, the fetch and the merge, instead of having both of those triggered at the same time.

---

### Gory is right

by [Jen Simmons](#) on Mon, 2011-01-31 12:26

Thanks for this clear write-up, Randy. I was still trying to understand exactly what was happening on this project — and this helps. We had a team of smart people who were all trying their best, yet day after day we'd awake up to discover someone had wiped out recent work. It was horrible. And about as useful as not having a version control system at all.

After this gory experience, I do worry a bit about Drupal.org projects. Anyone with commit access to a project in Git cannot only add new things or intentionally remove code, but they can delete things without knowing they deleted them (without meaning to) and *wipe out the history of a project...* it's a frightening recipe for disaster. We are all used to believing that no matter what happens, the version control system has a history, and you can roll back. In the situation Randy is describing, there was no rolling back. Code was wiped out, permanently. Gone. As if it had never been written. More than once, I did a pull, and watched in horror as my own local copy of hours of

work was snatched off my hard drive and destroyed. I got into the habit of making another new a copy of my codebase locally before each pull. :(

I'm glad Randy was involved with this story, and pulled Sam Boyer into the tale. I'm certainly going to be much more careful in giving out commit access to the Drupal projects I'm maintaining once we switch to Git. It's no longer a matter of making sure you trust the people who have access to the project. It's a matter of trusting their Git ninja skills, and that they understand what Randy's describing above, and know how to prevent it. I certainly didn't a few months ago. I think the short lesson is that if you are making a Git push, and wonder why your tool is going to push files that you haven't edited — *go ahead and push all of the files anyway!* Do not decline the chance to push files that you are not working on. You might be the one and only person with the correct copy of the code at that moment, and if you don't push it, it could be lost. The tough part is that this situation usually happens when you are confused about what's going on, deep in a mess of a failed merge, and it seems like a good idea to not mess with files you didn't edit lately anyway... but the instinct to "leave them alone" is wrong. You might be destroying them if you do that.

I look forward to your Part 2 tutorial, Randy. And thank you for all the things you are doing to teach us best practices using Git! I also love Git, and much prefer it to SVN and especially CVS. I'm glad Drupal.org is switching to Git, and hope eventually we can have some GitHub-like tools to help us easily understand how things work as we share our code.

## History was not actually lost

by rfay on Mon, 2011-01-31 12:34

The reality is in the scenario described, neither history nor code was lost, it was just a super pain to get back. It was still there in the repository, but had been "merged away". But it was too much trouble to solve.

## git fsck to find lost commits

by Adam DiCarlo on Mon, 2011-01-31 15:22

You can try git fsck to find the missing commits (well, patches) and try to reapply them. Yeah, it's a real pain, and it seems the older version of git I'm currently using (1.6.3.3) returns a lot of "false negatives" with git fsck (so even more time consuming).

## The repository was not corrupted

by rfay on Mon, 2011-01-31 16:16

Unless I misunderstand, git fsck is for a corrupted repo. The situation I'm describing doesn't have anything to do with a corrupted repo. It's a situation where a committer accidentally merges away a bunch of valid commits. The commits are still in the repo, but not in the HEAD version.

## SVN / CVS would have been worse

by grendzy on Mon, 2011-01-31 16:02

Sorry to hear you had such a painful experience - but I don't think it's completely fair to blame Git for this misadventure. I totally disagree with your conclusion that git is a "recipe for disaster" for drupal.org projects.

While it's true that another committer can mess up the code, this risk is the same in SVN and CVS as well. And Git gives teams many powerful tools that I expect will make drupal.org projects *more* productive.

Regarding the first issue, yes --force is destructive. And certainly, this error is confusing:

```
! [rejected]      master -> master (non-fast-forward)
error: failed to push some refs to 'stuff'
To prevent you from losing history, non-fast-forward updates were rejected
Merge the remote changes before pushing again. See the 'Note about
fast-forwards' section of 'git push --help' for details.
```

However, if we take git's advice and look at the help page, it does not suggest using --force as a remedy. Rather we see this warning:

In such a case, and only if you are certain that nobody in the meantime fetched your earlier commit A (and started building on top of it), you can run "git push --force" to overwrite it. In other words, "git push --force" is a method *reserved for a case where you do mean to lose history*.

The second issue - that of "Merging Without Understanding" - again is present in CVS and SVN as well. In fact every time you type "svn up" you are performing a merge, with the potential for conflicts. Git handles conflicts better than most, but they do occur. Rebase - while sometimes appropriate - is not the cure-all.

The best way to avoid merge conflicts is to 1) communicate with your team members about what you are working on, and 2) Avoid

committing "meow", that is unnecessary noise in your commits (such as formatting changes made by your editor).

---

## Whoa!

by rfay on Mon, 2011-01-31 16:14

I never said anything about git being a "recipe for disaster" on Drupal projects! I'm currently in the process of trying to help educate people about the git rollout and I'm a huge git fan. Wow. Where did that come from? I'm warning about the problems with a very specific workflow.

And no, this problem would not have happened in SVN or CVS, because only the conflicted items (not the successfully merged items) would be the responsibility of the random developer. That's basically what this article is about: In the git git, the developer (every developer) is for a time responsible for everything, even things they may not have touched. The problem here is triggered by a merge conflict, but the disaster is in the successful merge (that can get lost in this catastrophic situation).

---

## "recipe for disaster" was

by grendzy on Mon, 2011-01-31 16:46

"recipe for disaster" was quoted from Jen Simmons' comment.

I'm not sure I understand how git would ever ask you to resolve conflicts in files you didn't modify - unless perhaps you are tracking multiple remotes? Can you provide more detail on what might cause this?

---

## I didn't mean that moving

by Jen Simmons on Mon, 2011-01-31 16:56

I didn't mean that moving Drupal.org to Git is a recipe for disaster.

This is the recipe for disaster:

people with commit access to a project

+

doing the thing Randy described

=

"deleting things without knowing they deleted them (without meaning to)"

Believe me, it's painful. And if it were to happen to a precious Drupal project, like, say Views — it would be really bad.

(And yes, technically there is some evidence of the code's former existence in the history — but not in form that's very useful. I spent a day trying to recover code that way, and gave up. It was faster to just rewrite it all.)

Do I think we should not use git for drupal.org projects then? Oh heck no. I'm very happy about the git migration, and can't wait until it's done.

We just need to talk about this — there is a way for people to wipe out work much more easily than ever before. Personally, I am going to be more careful about who I give commit access to the projects I maintain.

My point is that under CVS, you had to intentionally remove something. With Git, you can remove work by accident. It's true. I saw it happen over and over.

---

## Very unlikely to happen without several committers

by rfay on Mon, 2011-01-31 16:59

I should mention that this exact scenario is quite unlikely unless you have several committers and are using the merge workflow.

Most projects in the Drupal world have few committers. I would imagine many will start using the gatekeeper model, where one branch is maintained by one person, pulling in what other people do.

---

## unlikely....

by Andyb on Tue, 2012-10-09 09:28

Very unlikely really means "going to happen next week for sure". As long as you have 2 committers you have the chance of this happening, and once it happens you'll be unhappy. Never, ever trust your source code to "well, it might never happen".

---

## How the merge workflow works...

by rfay on Mon, 2011-01-31 16:56

When several people are committing... and I pull their work into mine, by default an automatic merge takes place. I have to push that merge commit back and not mess it up, or things go bad.

The situation I'm describing happened when a pull with its automatic merge happened. Most of the merge (as usually happens) was successful. But one thing conflicted. The conflict was resolved. But in this case the developer didn't commit all the other things (that were successfully merged and automatically added to the index.). He only committed the thing that he had resolved (that he understood). He didn't know that he was responsible for all that other merged stuff that had been successfully merged and added to his index. That's the story of this disaster.

---

### The workflow had multiple issues

by Mikko Rantalainen on Fri, 2011-10-28 00:15

"The conflict was resolved. But in this case the developer didn't commit all the other things (that were successfully merged and automatically added to the index.). He only committed the thing that he had resolved (that he understood). He didn't know that he was responsible for all that other merged stuff that had been successfully merged and added to his index. That's the story of this disaster."

Clearly those developers did not understand the merging at all. That's issue number 1. However, I think that the greater issue is issue number 2, which is sharing write access to a single repository. The issue number 2 is made even more problematic because of issue number 1.

If you use \*distributed\* version control you should take advantage of the distributed part fully. That means, no shared write access to a single repository. The only correct way is to have one developer that is responsible for his repository and other developers send pull requests to him. And other developers should consider this key developer as main integration point. All other developers only get read access to this repository. Of course, it makes sense to have one or two "hot stand-by" developers for this key developer in case something bad happened to this key developer. The "hot stand-by" may be implemented with shared write access to a single repository where the "hot stand-by" only uses his write access if the key developer is unavailable. Or the "hot stand-by" could be just another developer with his own repo with only his write access and the "hot stand-by" would be implemented as clear communication to others that he is the "hot stand-by" in case the key developer is unavailable.

Check out the linux kernel development model; it works for a project that has hundreds of commits a day, so perhaps it could possibly work for a smaller project, too. And that development model is not too hard to follow for a smaller project either.

---

### what?

by erik on Fri, 2012-03-30 23:22

i think you have a bunch of totally false information here. can you clarify if you know what you're doing, or if you are beginners? i am wondering if i need to worry about your scenario #2 -- but as far as i can tell, you're totally wrong.

specifically, this sounds crazy:

"In the git git, the developer (every developer) is for a time responsible for everything, even things they may not have touched."

and so does jen's claim:

"You might be the one and only person with the correct copy of the code at that moment, and if you don't push it, it could be lost."

unless you describe how these are true claims, i warn readers that this is complete bull.

you say the missing work was right there in the history, but in some kind of "useless format"? what the heck do you mean? why could you not simply check out the commit before the merge? (similar to the comment below you didn't respond to: "You could have reset the HEAD to the commit just before the failed merge")

i do not understand the problem merge -- if they unchecked files "they hadn't touched," how is it that those files would have auto-merge results in them that would then be lost? if they had no edits, there would have been nothing to merge. when they pushed, nothing would happen to those files. (this point is similar to the comment above you didn't respond to: "I'm not sure I understand how git would ever ask you to resolve conflicts in files you didn't modify")

you aren't very clear if these are files he "hadn't touched" -- you say "files he was not involved in" -- so maybe you mean that they weren't involved in the conflict he had just manually resolved, but did include auto merges of his edits? that's the only thing that makes sense, because they wouldn't show up in the list of files to commit otherwise. but i still don't understand --

it's totally fine to leave them out of the commit -- his own changes would still be right there in his local working copy, and others' commits (that you say went missing) would still be the latest version of those files in the repo. exactly what caused someone else's commits to go missing, or his own uncommitted changes to disappear from his own working copy?

exactly what part of this has to do with inappropriately using an SVN mental model and not understanding git? in either case, someone can do a local merge, followed by a commit/push that doesn't include all changes. the uncommitted changes will still be present in their local working copy, and everyone else's commits will still be in HEAD. and in any case, it is trivial to checkout a revision from before the problem. what am i missing?

---

## This

by Johan on Sun, 2013-08-11 15:48

Maybe a little late reply. But here an example.

Suppose you are a git-ignorant developer

You edit file A

Commit

They edit file A, B and C

Commit - pushed

You pull their changes with TortoiseGit -> Conflict

When you try to commit you see file A conflicted, and file B and C changed

You uncheck B and C for commit because you do not remember those are your changes. Which is correct because those are their changes, but they still need to be in the merge-commit you are going to make.

You commit and push.

Now the change to B and C is undone in this merge commit. This is not correct. (Yes the changes are still in the history, but they shouldn't have been undone by the merge commit.)

---

## Exactly what happened

by rfay on Sun, 2013-08-11 18:24

You have it nailed.

---

## To me this seems to be more

by Christian Afonso on Mon, 2013-10-28 14:14

To me this seems to be more of a problem of TortoiseGit than of git itself. Sure, it is possible to do this in raw git, but it takes an amount of active work (manually removing the "unconflicting" changes and merge markers, or hard resetting the files) that you could equally do in svn ("merging" by removing the incoming changes in the diff editor). If it's as easy as just quickly unchecking a row of checkboxes, the interface is doing something wrong (akin to a big "delete all" button labeled "easy solution").

---

## I actually disagree: This is

by rfay on Mon, 2013-10-28 14:26

I actually disagree: This is a result of the design decision in git where the *committer* has control of the whole repo at once, unlike svn. A poorly informed committer can easily cause global and very ugly changes. Tortoisegit is only one way to do this.

---

## Why unlike svn?

by Johan on Mon, 2013-10-28 14:39

In SVN, especially TortoiseSVN, you can do the same thing.

Simply merge a revision range from another branch. Commit, resolve conflicts in the commit dialog, uncheck all other changed files (except the property change on the directory) and commit.

Now SVN stores the revision range as being merged while it is actually not merged at all.

---

## The committer owns the whole repo

by Mikko Rantalainen on Tue, 2013-10-29 04:30

You do realize that the git is a Distributed Version Control System (DVCS)? Every time you do a commit you do it in `_your_` repo and, of course, you have total control of the whole repo because it's `_yours_`.

With SVN the only way to avoid getting trash into the repo is to prevent/avoid committing at all, which is exactly the reason not to use SVN for anything. With distributed version control system `_you_` can always commit to your own repo and it's up to the maintainer (or canonical repo owner or whatever you have) to check suggested patches or merges before accepting those changes into his or her own repo. If the maintainer (or canonical repo owner) does not understand how to do that, he or she should learn about it before taking that job.

You `_will_` have major problems if you pretend to have distributed version control system where everybody does commit at will but in the same time your "distributed" repo is in fact a single shared resource on a central server. If your team is good enough, you `_can_` have shared central repo where everybody in the team is allowed to accept/merge/push patches into the canonical repo but in that case, you just have shared ownership of the canonical repo. However, before jumping to that style, I'd seriously recommend using single owner model, instead; that model `_does_` work for projects as large as the Linux kernel (code size increasing around 1-3 million lines per year, roughly 8k patches per release every 8-12 weeks and code from around 1000 individual programmers per release: <http://royal.pingdom.com/2012/04/16/linux-kernel-development-numbers/>). It will work with lesser projects, too.

---

## I actually agree with Mr

by [Caue Rego](#) on Wed, 2013-12-04 09:54

I actually agree with Mr Afonso. Sure, TortoiseGit is only one way to do this, but it encourages it, by leaving the responsibility of showing the files to Windows Explorer. I don't think anyone using bash, Git Extensions or SourceTree, to cite a few examples, would make the mistake you described here, rfay.

But I think your disagreement goes much deeper than this... Git has the philosophy of a powerful weapon without a lock or restrictions. It's up to the user to be careful. I can see this pattern in many places and it's hard to argue in favor of adding restrictions there or not. I, like I think Afonso would as well, vow for the art of learning to dome the beast, rather than putting it in a jail or even a zoo - but I can also appreciate a zoo from times to times.

---

## Thank you for this post, and

by [Scott](#) on Tue, 2011-02-01 05:44

Thank you for this post, and to all of you adding comments to it.

I look forward to more posts helping Git users avoid disaster.

I love Git, but there is clearly much I do not get yet (noting I'm still fairly new to version control anyway).

While a bit off-topic (so I don't intend on receiving a reply here about it), my Git oddity came when I first cloned a repository, made a commit locally, and then pushed the result back to the remote.

My push was rejected. The solution was to add a branch to the remote, switch out of master to that branch, and then push from my local clone (which magically updates the master branch on the remote without rejection).

My point is there is nothing intuitive about that, even though the reason for the rejection (which I read somewhere, and -- sigh -- now forgot) is valid.

Please continue communicating with us about how to properly use Git, and thank you for whatever disaster you prevent in my Git experience.

---

## #drupal-gitsupport

by [rfay](#) on Tue, 2011-02-01 07:02

I think you'll be able to find some help on #drupal-gitsupport, and encourage you to ask there.

-Randy

---

## Bare repository?

by [Izkata](#) on Sun, 2014-06-22 01:28

It sounds like you didn't clone a bare repository, but a full one. Which means there could have been local changes where you were trying to push to.



That's why switching it off of master worked - only the branch HEAD is pointing at is capable of having local changes. By pointing HEAD to a second branch, master was then safe to push to.

## Our git experiences

by [J-P](#) on Thu, 2011-02-03 09:04

Great summary of the pitfalls, Randy. Like any advanced magic, you don't want it going off in your hand; especially if you can do a pull request and have it go off in someone else's ;) That denyNonFastForwards trick is lovely: but doesn't --system mean it stores it systemwide, so you don't need to do it on a particular repository?

At Torchbox we're about 80-90% migrated from subversion to git (including historical projects), and all our new projects are in git. I've found it lovely to work with, but pretty arcane at first. Certainly being able to handle other people's changesets like they were objects in code has meant we've been able to deploy new changes across many feature branches, very quickly. But the D in DVCS is the kicker, and I don't think people from an svn background (me included) appreciate at first how it changes things.

One other disaster people like might to avoid: **pulling changes into a local repository, while you also have locally uncommitted changes**. Always do a git commit (or a git stash) before a git pull, because when you have three things to reconcile - local edits, local HEAD plus changesets, remote changesets - then it can get very nasty indeed.

With older versions of the git client I think you simply could not back out from this situation through version control (and file edits) alone, and instead had to copy all your files somewhere (even git stash didn't help), hard reset, and then copy them back. On that note, I'd also advise people to always use the most up-to-date point version of git - so 1.7.x at the moment.

Otherwise... anyone any success with using [git flow](#) to avoid developer-to-developer conflicts?

## Understand rebase -- the solution to all your problems

by [craigslis48335](#) on Thu, 2011-02-03 13:28

Been there.. seen that. And yet, I'd say GIT let me recover from similar situations far more gracefully than SVN/CVS. On some projects using SVN, I've seen the formal institution of merge time-windows - where a particular order of merging different developer's work was decided by release manager & formal handover of control to 'master' branch was done in that order. In such scenarios, the person who had control of merge was responsible for (a) not introducing regressions (b) and the end getting this certified .

When working with GIT, I'd say the general rule of thumb is try to structure your workflow similar to the gated 'pull' workflow of Linux kernel development. By this, I mean try to have smaller groups of developers- that work on related files- that merge between themselves. Coordination and MERGING between multiple such groups should be done by someone more experienced with git AND \_more importantly\_ more knowledgeable about your codebase. This is really important to be able to spot & resolve merge conflicts.

((1))

If developers in a group are newbies/you don't trust them with not fucking up a merge : this almost always implies that these developer is NOT working on a multiple independent tasks or that he's making large number of changes...: In this situation, don't let that developer push to common repository. Make that person do this (1) Fetch from authoritative repo, (2) rebase or merge changes (3) TEST locally & if applicable, pass a test suite. Only after the developer demonstrates to have undertaken ,these steps , then (a) a designated person - presumably a more experienced developer - 'pull's from the developer. If the original developer had done his local merge well, then the result of pull by merge developer should be CLEAN - and should apply without conflicts. If conflicts exist : either the inexperienced dev didn't do a clean merge OR upstream changed. If upstream has changed, then the lead dev should be able to handle the conflict resolution.

((2)) If required, using patch files works perfectly with GIT, so use this functionality. Also, this is preferable to emailing zip files containing modified files to each other (yes, ugly , but lot of people do this to share work while trying to get around merge/commit issues)

((3)) Educate people, possibly even evangelize about the power of and proper usage of 'rebase' -- especially squashing multiple commits before publishing. I can't express how much this helps in keeping history cleaner or reverting to particular version easy.

((4)) Above everything, emphasize the importance of TEST after merge .

## Github method working well at sony.

by [frankcarey](#) on Fri, 2011-02-04 14:47

I'm planning a blog post about this myself advocating the github model. We actually use both methods in our infrastructure now at sony music, where we have a behemoth platform, and almost 300 sites like michaeljackson.com and christinaaguilera.com running on basically one multisite. We use the merge method for site folders (which are their own repos in our system), which is fine because it's basically theme work only and it's done by one or two themers. For the platform (drupal core + sites/all) we use the github (everyone has their own forks and does pull requests) method.

We have a committer team who have "push" permissions on the "upstream" repo, and they do the code reviews and provide feedback



and questions. After the code is ready and tested, it gets merged back into HEAD. We've been using the github interface for this workflow, and it's been really great, especially when you have external devs or contractors or otherwise have a large team and want to ensure the quality of commits. Stay tuned for details, but thanks to rfay for getting the options out there. If you've never met Randy, he's probably one of the nicest people you'll meet in the drupal community (and it's generally a really nice community!)

---

## Understand rebase -- the solution to all your problems

by craigslist48335 on Thu, 2011-02-03 13:35

Been there.. seen that. And yet, I'd say GIT let me recover from similar situations far more gracefully than SVN/CVS. On some projects using SVN, I've seen the formal institution of merge time-windows - where a particular order of merging different developer's work was decided by release manager & formal handover of control to 'master' branch was done in that order. In such scenarios, the person who had control of merge was responsible for (a) not introducing regressions (b) and the end getting this certified .

When working with GIT, I'd say the general rule of thumb is try to structure your workflow similar to the gated 'pull' workflow of Linux kernel development. By this, I mean try to have smaller groups of developers- that work on related files- that merge between themselves. Coordination and MERGING between multiple such groups should be done by someone more experienced with git AND \_more importantly\_ more knowledgeable about your codebase. This is really important to be able to spot & resolve merge conflicts.

((1))

If developers in a group are newbies/you don't trust them with not fucking up a merge : this almost always implies that these developer is NOT working on a multiple independent tasks or that he's making large number of changes...: In this situation, don't let that developer push to common repository. Make that person do this (1) Fetch from authoritative repo, (2) rebase or merge changes (3) TEST locally & if applicable, pass a test suite. Only after the developer demonstrates to have undertaken these steps , then (a) a designated person - presumably a more experienced developer - 'pull's from the developer. If the original developer had done his local merge well, then the result of pull by merge developer should be CLEAN - and should apply without conflicts. If conflicts exist : either the inexperienced dev didn't do a clean merge OR upstream changed. If upstream has changed, then the lead dev should be able to handle the conflict resolution.

((2)) If required, using patch files works perfectly with GIT, so use this functionality. Also, this is preferable to emailing zip files containing modified files to each other (yes, ugly , but lot of people do this to share work while trying to get around merge/commit issues)


((3)) Educate people, possibly even evangelize about the power of and proper usage of 'rebase' -- especially squashing multiple commits before publishing. I can't express how much this helps in keeping history cleaner or reverting to particular version easy.

((4)) Above everything, emphasize the importance of TEST after merge .

---

## I feel as if I must be missing something here...

by Kyralessa on Wed, 2011-03-09 12:54

 The result: All the commits by other people that had been done between this user's previous commit and this one were discarded.

Were they really *discarded*? That would imply that they had completely disappeared from Git's history, as though the checkins had never occurred.

---

## Still in Git's history... but...

by rfay on Wed, 2011-03-09 13:15

Yep, they're still in Git's history, but they might as well not be because the team was completely confused. Basically the merge commit that was committed back did *not* contain those changes, so they disappeared from this branch.

---

## I've been reading and

by Alex Beamish on Fri, 2011-10-28 14:39

I've been reading and enjoying your posts about git -- it's a challenging tool, but works well when you really understand the logic behind version control.

This case of the commits that didn't contain the changes shouldn't have been a problem -- it should have been possible to cherry pick those commits and thus restore them. Maybe a later comment will explain if that happened.

---

## Thanks - yeah, it wasn't

by rfay on Fri, 2011-10-28 15:01

Thanks - yeah, it wasn't *commits* that were lost in this case, it was a merge commit that subtly removed all the changes that

had been made by other commit. A merge commit has a lot of stuff in it, and things can go really wrong :-)

---

## Still confused

by Till on Thu, 2012-01-12 10:57

I still do not see the problem. If you found out fast, that the merge caused trouble and it's too late to reset, why were you not able to do a simple "get revert \$mergecommit" undoing any changes that were done at the merge (as well as the merge)? I see the part about such a commit causing trouble, but not the part about this trouble taking a long time to remedy.

If it was not obvious which commit caused the , a "git bisect" should have only taken a moment to find the correct one.


In my opinion, this is one of the nicest parts of git: It takes great care to preserve your work, even if somebody screws up (for example by messing up a merge). At worst you will have to dump their work from your branches, but then it's their problem to figure out how to get it merged again later (it won't be lost either though).

---

## TortoiseGit

by bumpaw on Thu, 2011-04-21 13:03

I was happy when I saw you write:

 # Many of the team members were using Tortoise Git, which works fine,

Most of the time all discussion of git focuses on using the command line, and I really avoid it when I can. I'm hoping there will be more git instruction pointed at TortoiseGit users. Looks like from this article that more instruction could have saved some headaches. Tutorials for it should grow now with the migration complete. My real hope is that you will become such a fan of TG that you will do some. You have a gift for instruction. :)

---

## !TortoiseGit

by Stephen C on Fri, 2011-06-03 16:59

Funny you give credit to TortoiseGit, I was actually going to comment to the inverse. Certainly the main issue here is lack of knowledge about git, for which there *is* a learning curve, no doubt. Dumping people into an environment they don't understand is the real recipe for disaster here. I blame TortoiseGit (after lack of developer knowledge) in this specific case for its ease in "checking off files" that changed but the developer didn't understand were necessary as part of their merge. In the command line, there is no such option to simply deselect a bunch of files from the commit. You'd have to do some serious ninja stuff to exclude files that were auto-merged and then you would certainly know what you were doing.

Having someone on the team who was more familiar with git could have made this 'disaster' far more easily recoverable. You could have reset the HEAD to the commit just before the failed merge, branched from there, and let people continue as though nothing had happened. Think about what you would do in the case that a developer simply deleted a bunch of files unintentionally and committed the deletions. Do you have continuous integration automatically testing your code base after each commit? You are writing tests for your code right? This would have immediately highlighted the problem commit for you. Trying to make do without that is another recipe for disaster.

---

## On many teams replacing /

by Eric on Fri, 2016-01-22 17:11

On many teams replacing / force pushing the main branch is a recipe for further drama and issues. It isn't practical for everyone.

Additionally, tests detecting this will only help you if your tests aren't in the same repo.. usually in TDD someone will write tests and code in the same branch and once it's merged they think "Oh, I'm done".

This isn't the case if someone else's mistaken merge comes in and removes both their test changes and their tests at once - test suite won't find that.

---

## Reading this again a year or so later...

by Kyrlessa on Sat, 2012-03-31 19:19

...I think your fundamental problem was TortoiseGit.

In our shop, I introduced Git to developers who were only familiar with TFS. The only graphical tool we've used is gitk. Everything else has been command-line only. It's not actually that difficult to do things in Git by the command line, and it prevents a GUI from working some

magic behind your back.

On another note, what the hell is wrong with you that you think anyone posting a comment here would know anything about Drupal?

---

## Drupal

by Dyz on Thu, 2012-04-19 08:14

Just to prove you wrong; I know about Drupal.

I've experienced the same problems with Git as described in this thread. Training developers to work with Git is very important to prevent the problems mentioned above. What I also did was create how-to descriptions with screen dumps for the developers because they are apparently too lazy to learn Git and they keep asking me for support (I got tired of googling for them).

---

## This problem is real

by method3000 on Fri, 2012-04-20 10:35

I don't know why so many people are reacting with aggression and denial to the description of the problem in #2. This problem absolutely happens and is HELL to detect and recover from. Yes, GUI interfaces make it easier to do for people who aren't familiar with the theory of Git. It's not corruption. It's something that TWO new developers have done in my organization.

What you're doing is saying, "the correct way to merge this code is to THROW AWAY every change besides the one merged file I'm aware of". Now Git knows that it should skip over the commit objects represented by those changes. Okay, well you can just revert the merge. No, because when you revert a merge you're saying, "I'm not ready to deal with these particular commit objects represented by this merge." If you try to re-merge from that branch you'll see that it actually skips over all the commits that were in the original merge. Subsequent merges from the same branch will skip over those commits, until you "revert the revert". Then you just pull back in that screwy merge resolution. What you have to do to fix the situation, short of rewinding the history (which you CANT always do) is checkout the refs that should be but are not currently HEAD and re-commit them. If the merge conflict occurred along with a lot of files this can be quite the undertaking.

---

## Correct this situation is not that hard

by Johan on Mon, 2013-10-28 15:11

To correct this situation, how I would do it, is simply, just redo it.

- Make sure you pushed your changes
- Reset hard to before the merge
- Redo the merge but now correctly
- Commit (this commit will get lost, but.. why not :) )
- reset mixed (not hard!) to the old merge commit
- commit the now correct tree state on top of the bad merge commit with the text 'corrected previous merge commit'
- pull (maybe resolve more conflicts)
- push again

done

---

## Ah but you forget: We're

by rfay on Mon, 2013-10-28 15:55

Ah but you forget: We're talking about problems caused by team members who have a very limited grasp of git. And you forget that this problem must be discovered and debugged first (by all the developers whose changes have magically been deleted).

The key here is to have a "gatekeeper" who does the merging. That will restrict the problem access to a trusted single point of contact.

---

## about caring for merges

by [Cauê Rego](#) on Wed, 2013-12-04 09:56

There's even more to worry about merging... <http://stackoverflow.com/questions/20380013/git-merge-strategies-spaces-...>

A merge that accuses no conflict can be one of the most dangerous kind.

That being said, I don't think rebase is better than merge. Michael's fetch & merge sounds a far more interesting kind of work around: in the devs minds. But, sometimes, we do need to add restraints and I also find it very fair to use a "gatekeeper", just like it's done in github.

In any case, awesome post, rfay. Good insights. :)

## I'm afraid recommending

by [staafi](#) on Fri, 2014-02-14 01:11

I'm afraid recommending "rebase" as a solution to non-fast-forward merge issues is rather meaningless. The rebased patches aren't applied on the same code they were originally created on so the with both commands, so the underlying operations are the same with both commands. Rebasing is useful for creating a simpler commit graph, it doesn't magically turn all merges into fast forwards.

---

## Linus Torvalds wrote about

by Mikko Rantalainen on Fri, 2014-02-14 02:25

Linus Torvalds wrote about rebasing stuff in 2009:

<https://www.mail-archive.com/dri-devel@lists.sourceforge.net/msg39091.html>

The most important part for this discussions is

It may also be worth noting that excessive 'git rebase' will not make things any cleaner: if you do too many rebases, it will just mean that all your old pre-rebase testing is now of dubious value. So by all means rebase your own work, but use *some* judgement in it.

In the end, I'd recommend rebase as the most simple solution. Just make "git tag BACKUP" before trying to rebase (and never rebase with a dirty work directory) and if things do not work out, running "git reset --hard BACKUP" will solve the problem. Only then you should try different merge strategies.

---

## Awesome hint for backing up!

by [Caue Rego](#) on Fri, 2014-02-14 13:35

Awesome hint for backing up! ;)

---

## bottom line..

by [sean](#) on Thu, 2014-09-04 17:02

at yahoo we were basically (recently) forced to migrate to git (svn previously), and this literally happened in about a week with little/no time for training. i have no idea what the motivation was and i certainly am not a huge fan of SVN...but that quick of a change for what seemed like draconian/arbitrary reasons has caused TONS of pain.

bottom line is..the learning curve is medium at best to hard at worst even for seasoned engineers (like mme). i also still dont think its the right source control system for us since we have 100s/1000s of engineers working on the same sets of files/repos. (but this is just my opinion)

its not that git doesnt give you more power....its that its simply much more cumbersome...as every dev has to be very careful with how they push/pull. and the funny thing the distributed model doesnt add any benefit to the developer so far as i can tell. even when doing everything right and what not. the only benefit i see for a dev is allowing you to keep track of changes you MAY want to make later (committing locally adding comments and what not) but thats never something i personally have ever needed. yes distributed makes "administration" a bit easier...but that to me comes with big costs for little gain.

i now basically keep 3 separate clones of everything just so i can ensure that a pull or push wont fail and i can quickly revert back if needed. i NEVER had to do that with CVS or SVN,

again im not arguing that SVN or CVS are better...im just saying that i see very little value add for devs along with cumbersome merge mgmt. even for git fanboys..and those who know what they are doing it takes more time. of course SVN and CVS have other issues many which are worse but they rarely (if ever) impact my btime day to day the way git does...

---

## It's a shame this seems to be a rite of passage for git users

by [Brion](#) on Thu, 2014-10-23 10:32

Rfay, I think for the most part this is a very reasoned and reflective post. I think your final conclusion of abandoning the merge workflow is a bit of a knee-jerk reaction to an admittedly horrible experience with it. But the recommendation to use rebase as a fix seems a bit hasty.

As has been pointed out in several replies there were several contributing factors to your disaster which I expect (hope?) you've learned from 3+ years on:

1. The devs weren't clear on the fundamental difference between a patch and a changeset (SVN vs. git)
2. TortoiseGit enabled dangerous behaviors too easily and without enough warning (possibly based on the assumption that TortoiseGit users

are very knowledgeable git users and should know what the tool is doing under the hood)

3. In the language of Larry Wall - There's more than one way to do it - there are more alternatives than just rebase or living with merge (ala 'git pull')

We have an organization with about 20 developers who work in a rapid development environment across over a dozen inter-connected and shared git projects (components of a larger system). Not all developers are equally well-versed with git, but we have a few resident self-taught experts (i.e. people who read Pro Git and make an active effort to expand their working knowledge about git to help others). However we have done several things that minimize and mitigate conflicts and collisions:

- Continuous integration (Jenkins) triggered on commits and with project dependencies such that a core project rebuild will cascade and rebuild dependent projects and email notifications of problems to the devs
- Bug tracking used with bug IDs in the commit message which can be identified programmatically (matches a simple regex) for reporting and linking commits to the bug tracking system
- Devs do the majority of work in a personal feature branch that follows a naming convention. Multiple devs working on related issues may collaborate on their own shared branch which periodically (when stable) is merged into the shared development/test/staging branch(es)
- Branches are only cut from master but can be updated directly from other feature branches they may need (due to parallel development) - git handles merges very well typically if rebase is not in heavy use (because that changes commit hashes and causes more conflict headaches when the same actual changes are re-merged)
- git stash is used as an essential tool for setting current work aside while merging in remote changes, and then git stash pop can be used to put the WIP back on the newly-updated branch. This is similar to git pull --rebase except that my commit hashes do not change and there's no risk to my in-flight work aside from remote changes that cause me to do some rework (which would have happened anyway)

I found <http://think-like-a-git.net> to be an invaluable resource along with <http://git-scm.com> for its reference (and of course the Pro Git book).

Here's to hoping you and others have better git experiences going forward!

## Rebasing is an antipattern

by PeterB on Wed, 2015-01-14 15:01

Rebase workflows seem like "clean" ways of working, but they're actually a mess. Every time you rebase, you're really creating a new branch, discarding all your commits, and adding new commits to the tail end of the new branch.

While this "prevents" a merge, what it does is shred your history. This is fine if you never intend to share your "private" work adhoc with any other developers, because if you share work that you rebase, anyone you've shared with will be forced to rebase and deal with any conflicts that come from it.

Merging is a much better workflow, and rebasing is common with git users because git makes merging harder than it needs to be. If you want to have a clean histories with merging, use `--no-ff` when merging branches.

Better yet, use Bazaar or Mercurial. They get it right out of the box and are a lot easier to use. See <http://duckrowing.com/2013/12/26/bzr-init-a-bazaar-tutorial/>.

## easy fix

by Tim Crall on Fri, 2015-03-13 13:48

I just recreated this whole problem.

The developer who did the merge and who had unchecked the files should have still had those changes as unstaged changes in their local repository. It should have been very easy for that person to just go ahead and add them and commit and push to fix this. There are of course a number of other ways to have fixed the problem as well. It was hardly the disaster it was portrayed as, as nothing was really lost, just slightly misplaced in easy-to-fix ways.

## Best Approach For Resolving

by Chas on Wed, 2015-04-22 07:44

You mentioned this is an easy fix, so I'm curious about the best approach for detecting and resolving this, especially where it didn't get caught immediately and tons of changes have been merged after this bad merge. If files have suddenly gone missing due to a bad merge (or merges) what can be done to find which files were lost and recover them back?

Drupal theme by [Kiwi Themes](#).