# The Git Parable

Johan Herland

<johanh@opera.com>

# The Git Parable

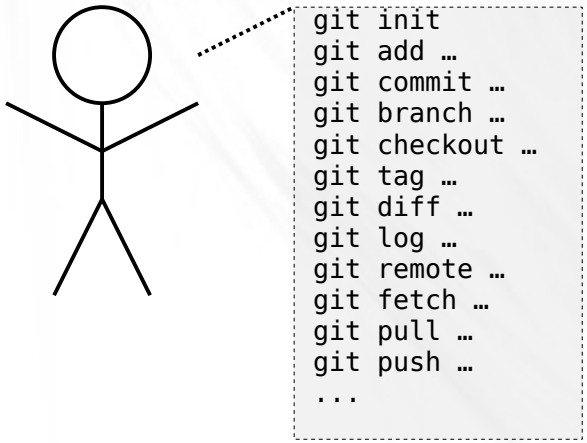
- Shamelessly stolen from Tom Preston-Werner
  http://tom.preston-werner.com/2009/05/19/the-git-parable.html
- I'm lazy...
- Also: Best introduction to Git I've found so far

# The Git Parable

- Git - simple & powerful

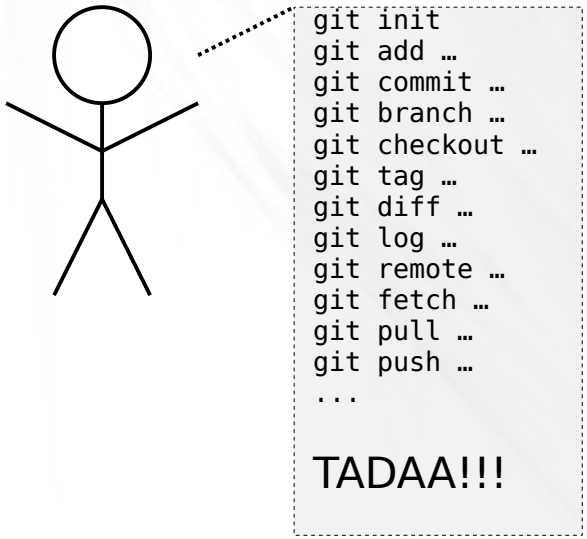
# The Git Parable

- Git - simple & powerful



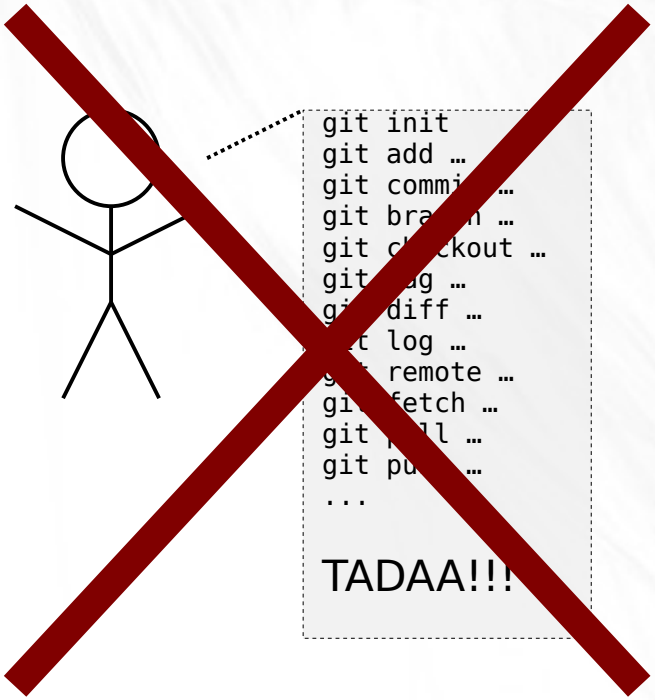
# The Git Parable

- Git - simple & powerful



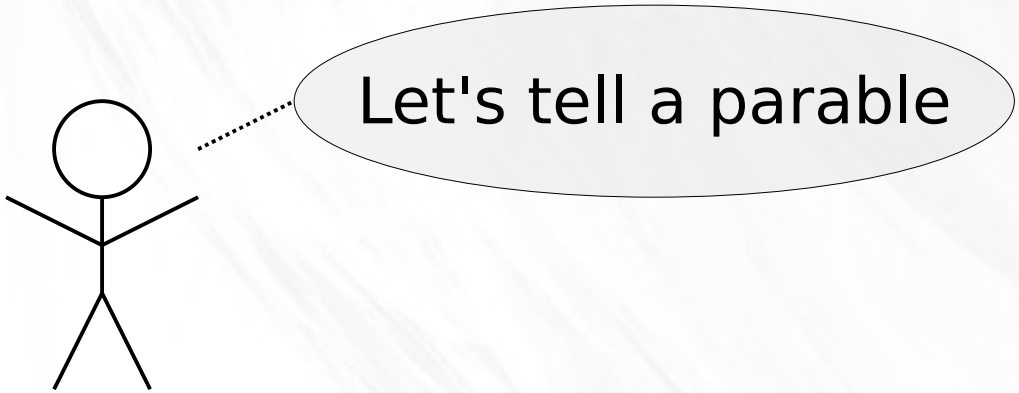
# The Git Parable

- Git - simple & powerful



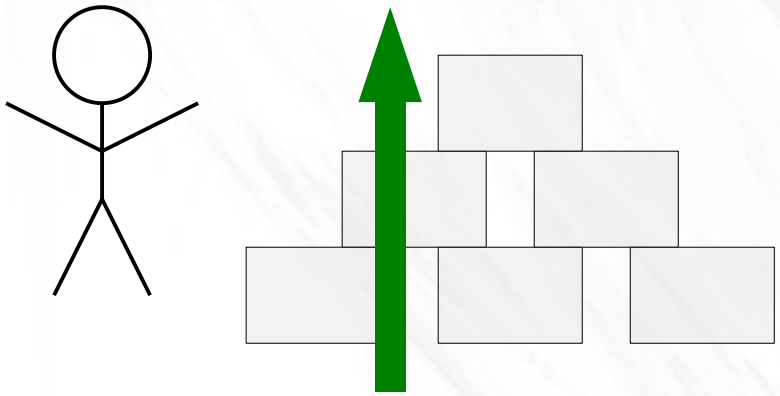
# The Git Parable

- Git - simple & powerful



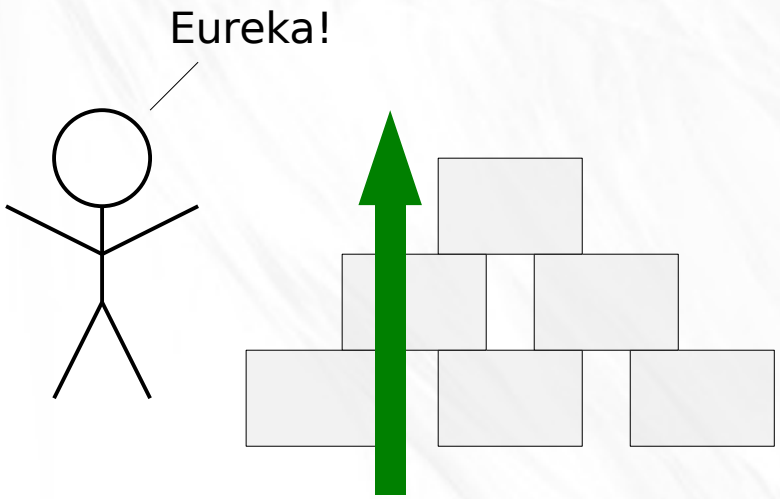
# The Git Parable

- Git - simple & powerful



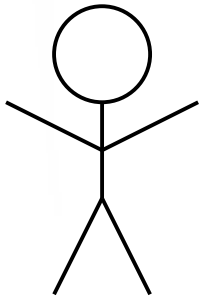
# The Git Parable

- Git - simple & powerful



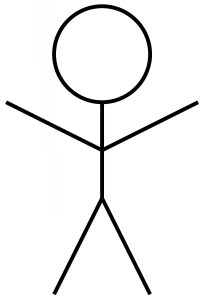
# The Parable

- A simple computer
  - A text editor
  - A few filesystem commands



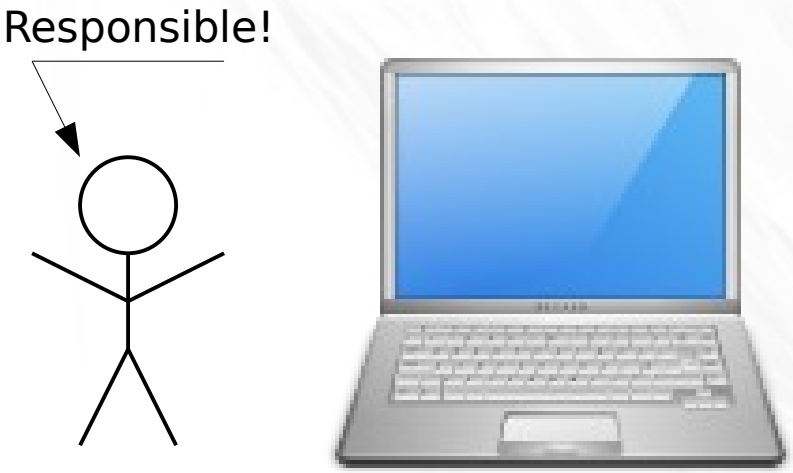
# The Parable

- Write a large software program



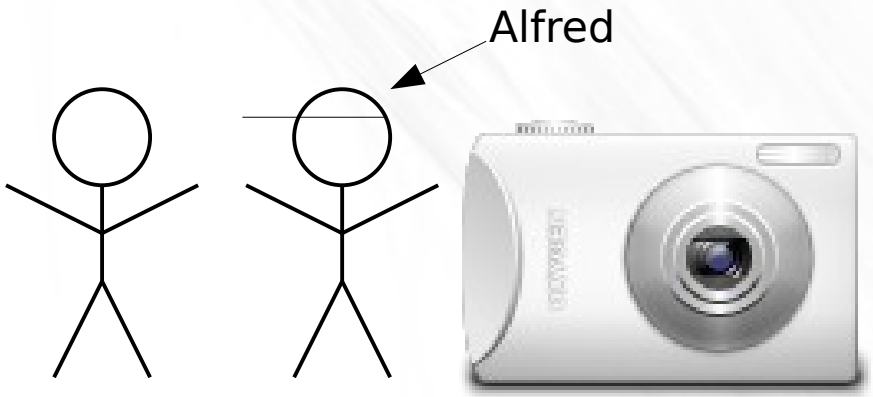
# The Parable

- Write a large software program
- Invent some method to keep track of versions
  - retrieve code that you changed/deleted



# Snapshots

- Alfred, the photographer

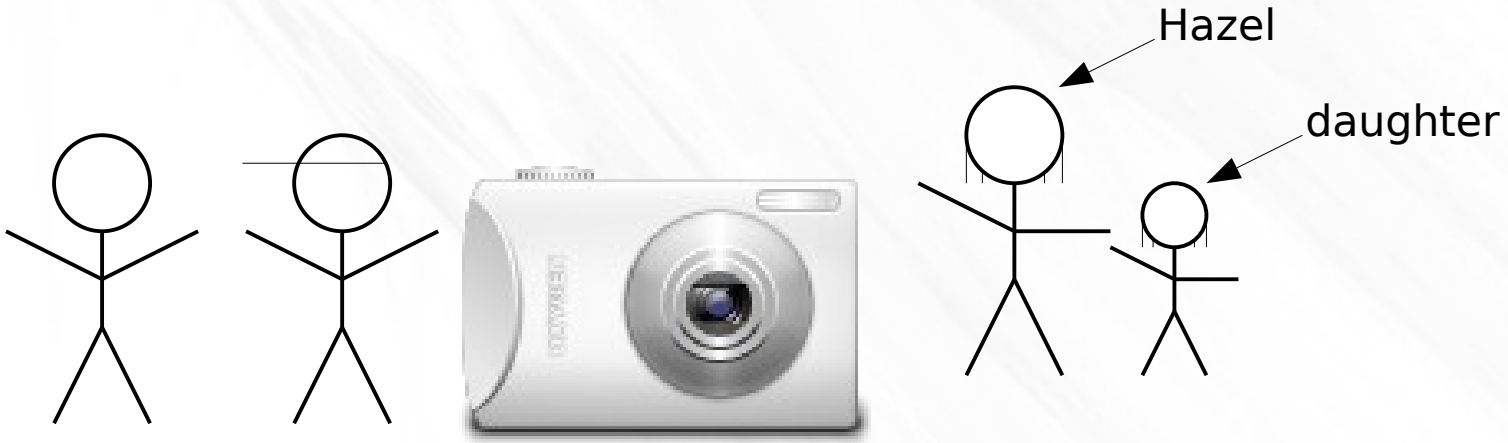


# Snapshots

- Alfred, the photographer

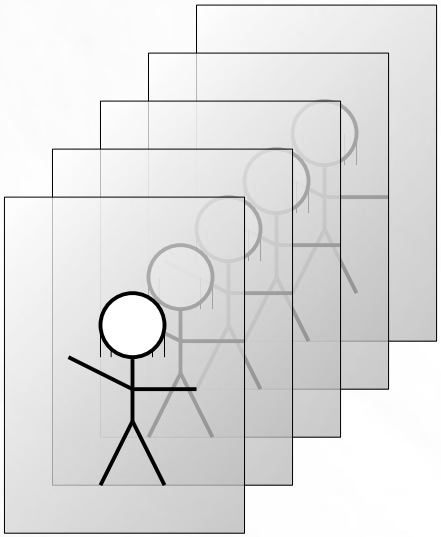
# Snapshots

- Alfred, the photographer
- Hazel and her daughter



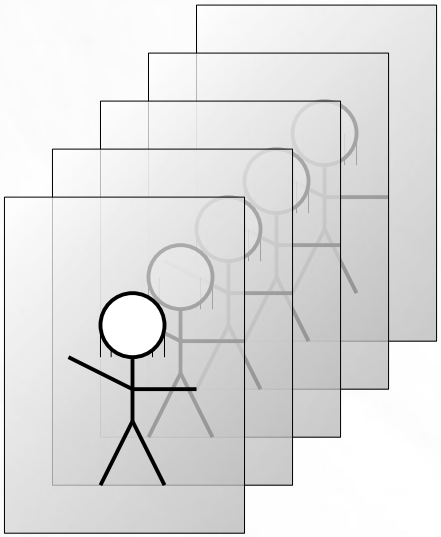
# Snapshots

- Alfred, the photographer
- Hazel and her daughter
  - Remember what the daughter was like at each different stage



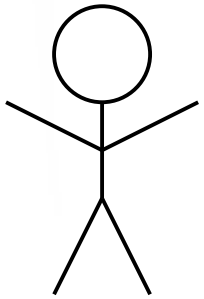
# Snapshots

- Alfred, the photographer
- Hazel and her daughter
  - Remember what the daughter was like at each different stage

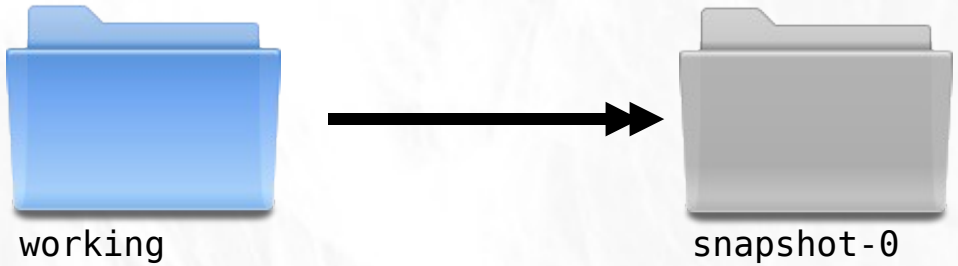


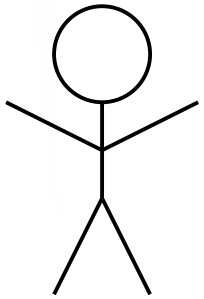
# Snapshots

working

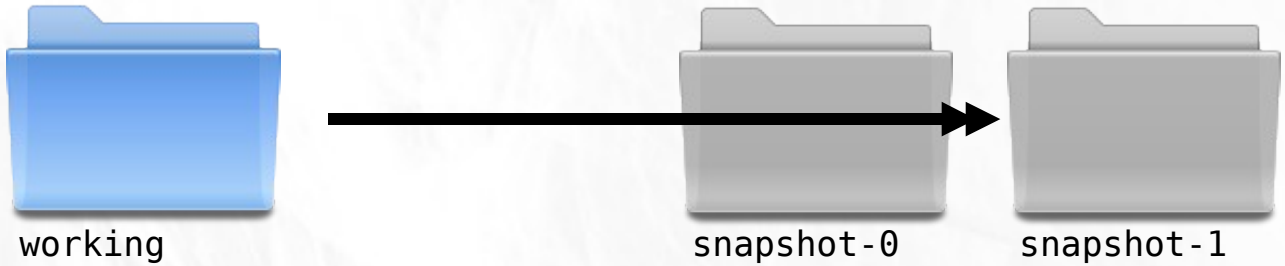


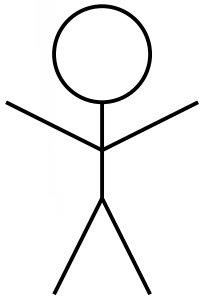
# Snapshots



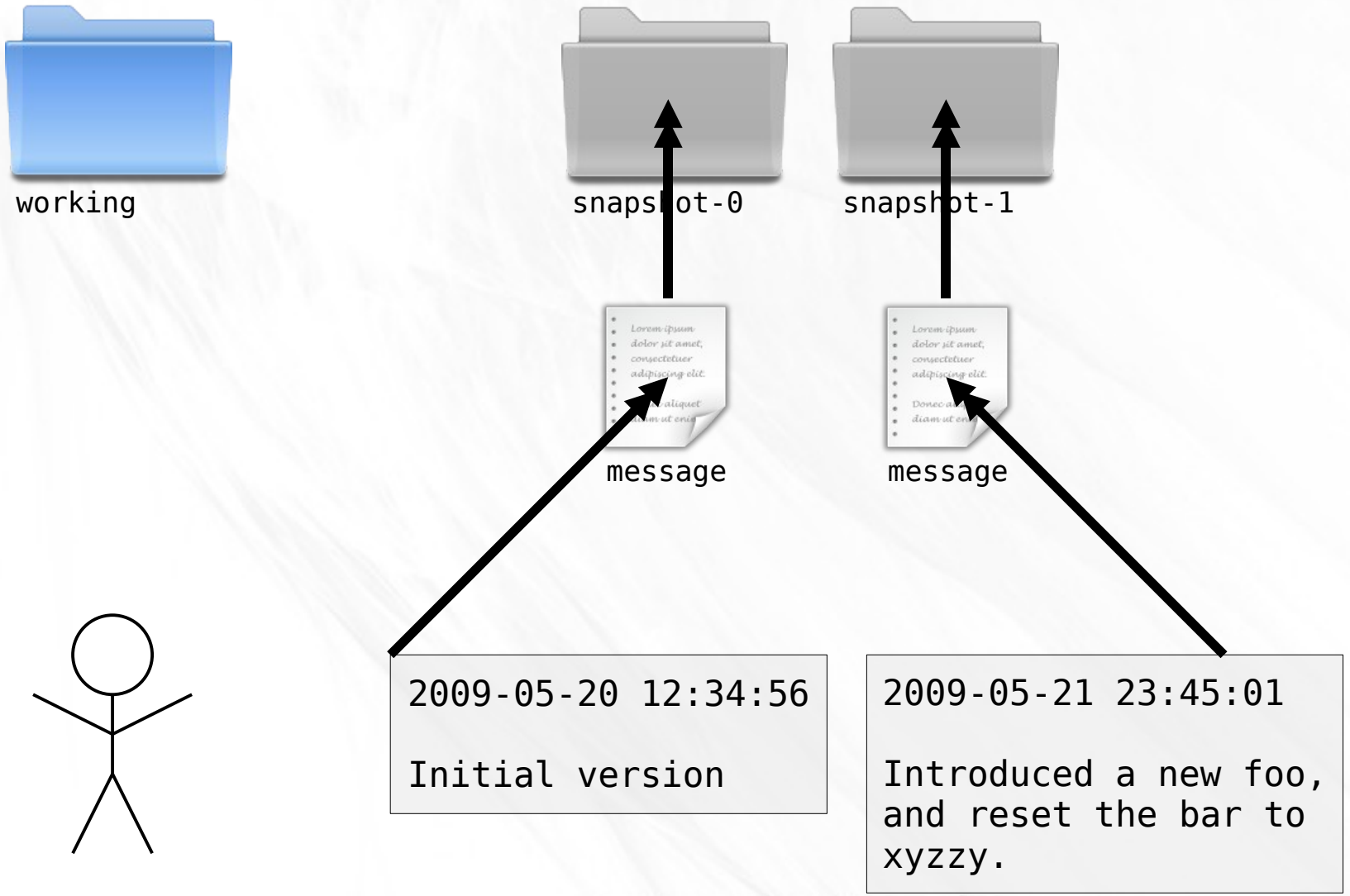



# Snapshots



# Snapshots


working
snapshot-0
snapshot-1
message
message
2009-05-20 12:34:56
Initial version
2009-05-21 23:45:01
Introduced a new foo, and reset the bar to xyzzy.

# Branches

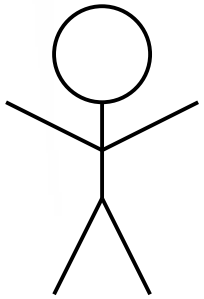
working

snapshot-0

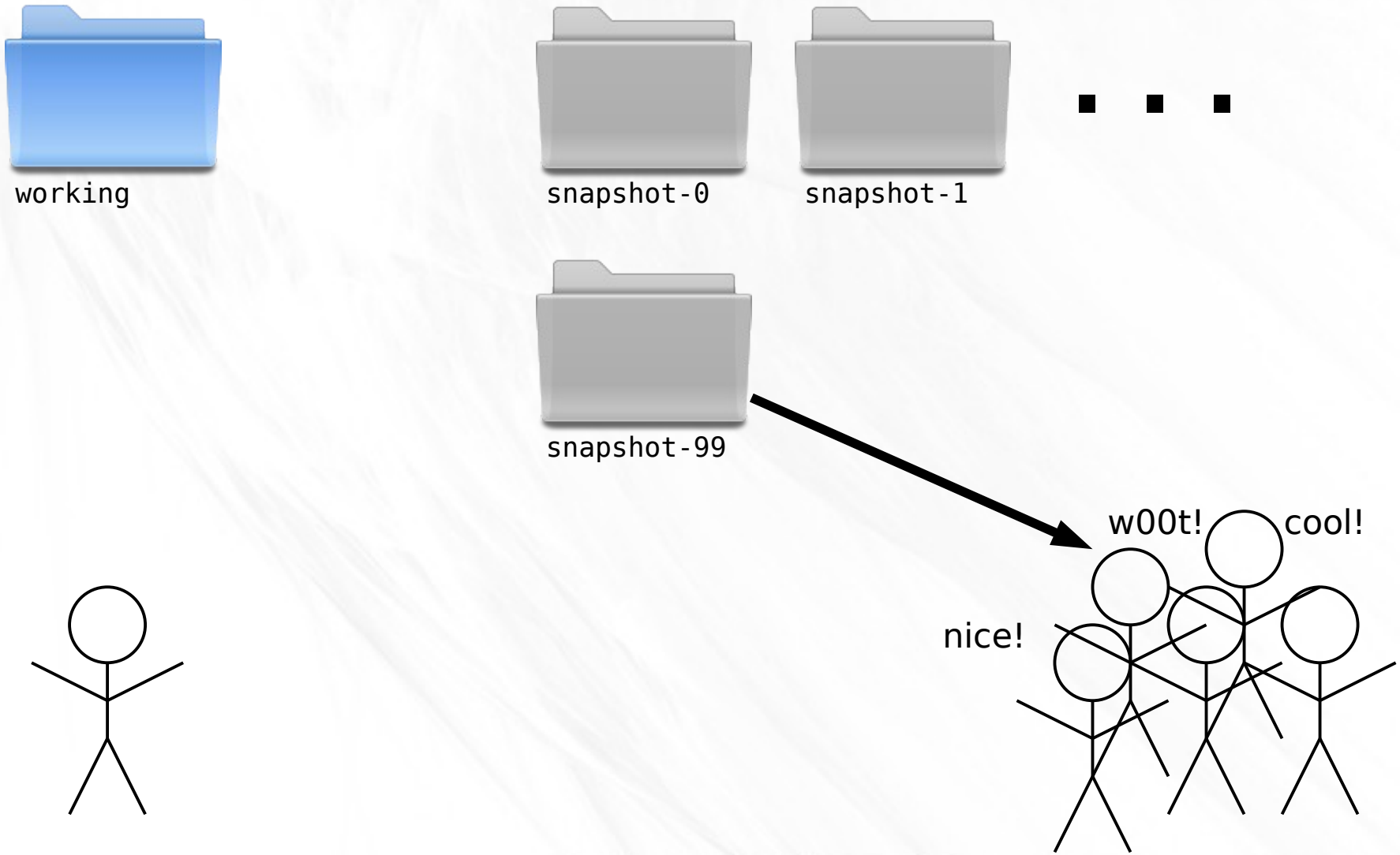
snapshot-1

. . .

snapshot-99



# Branches



# Branches

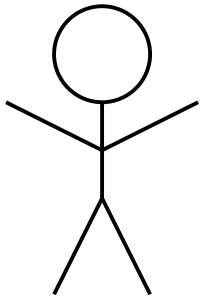
working

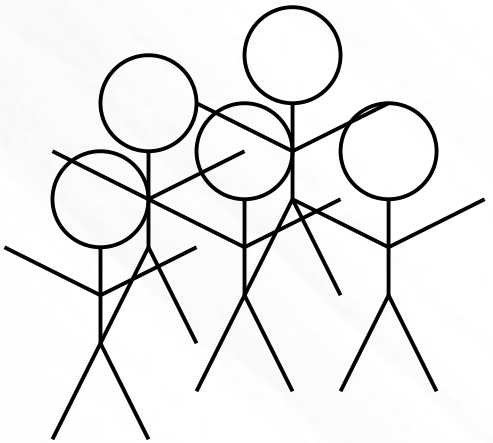
snapshot-0

snapshot-1

snapshot-99

snapshot-109





# Branches

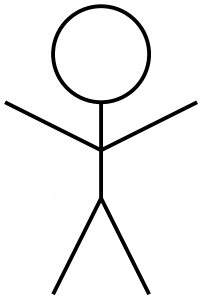
working

snapshot-0

snapshot-1

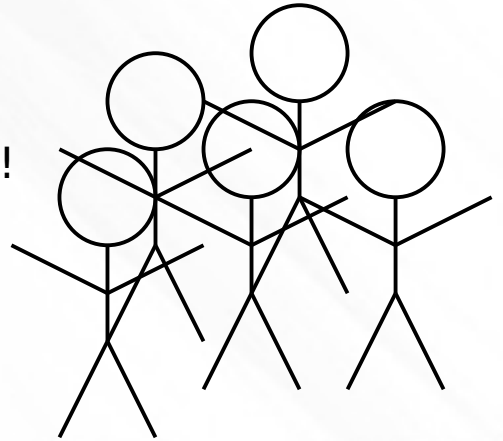
snapshot-99

snapshot-109

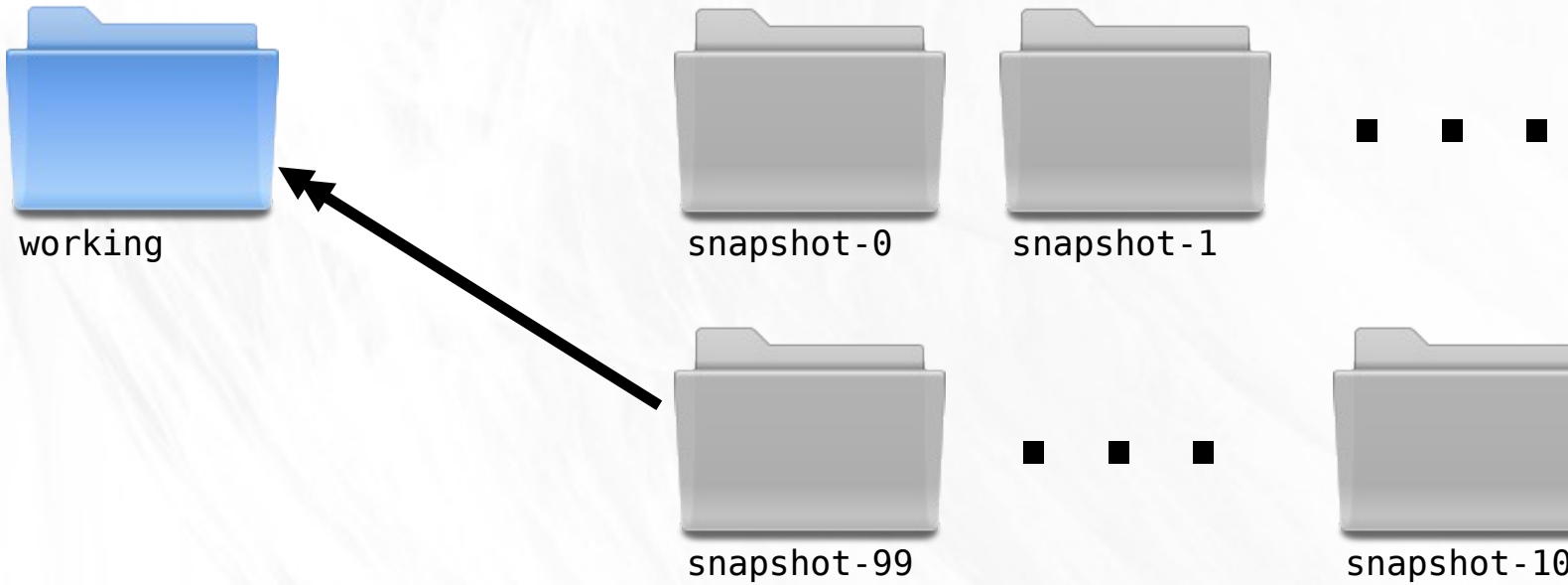


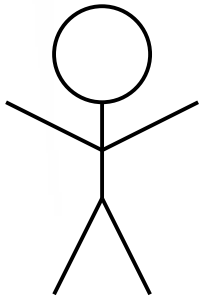
Does not work!

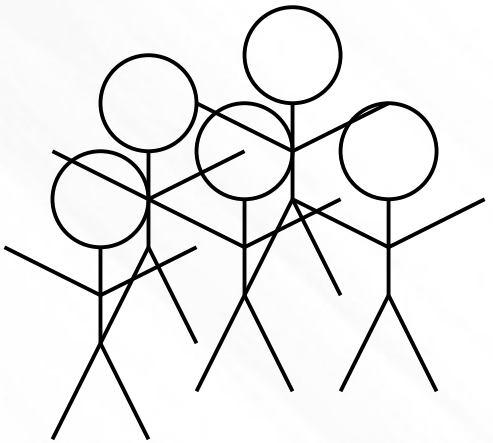
Boo!



# Branches





# Branches

working

snapshot-0

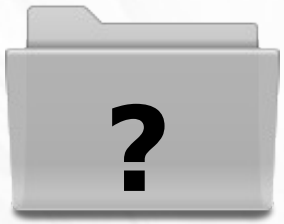
snapshot-1

. . .

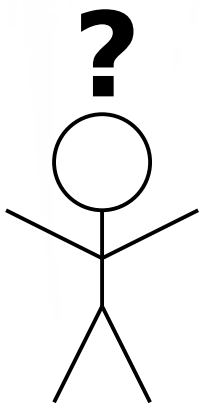
snapshot-99

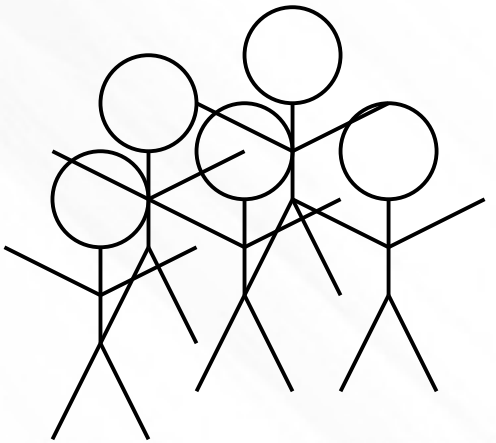
. . .

snapshot-109



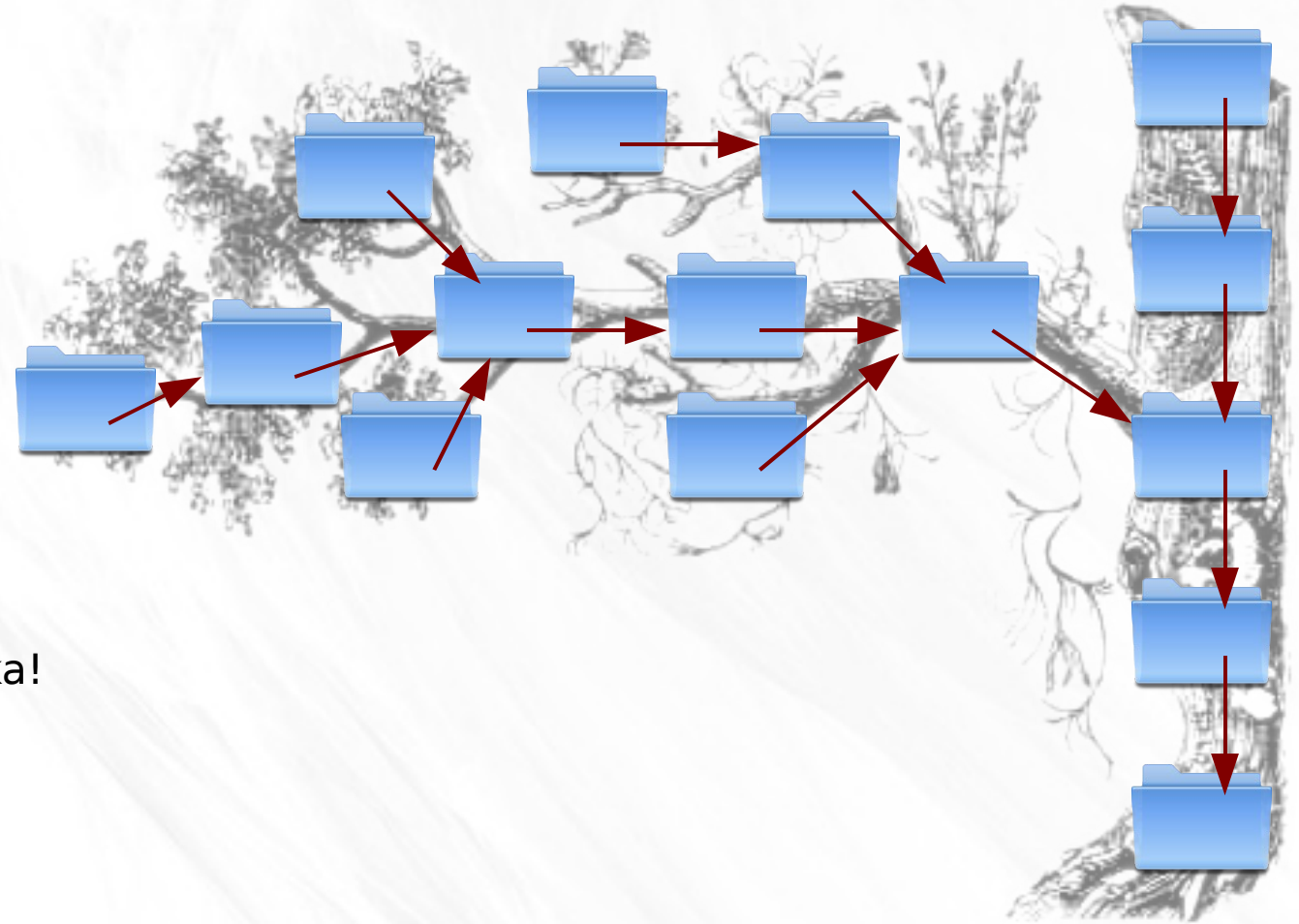

snapshot-110

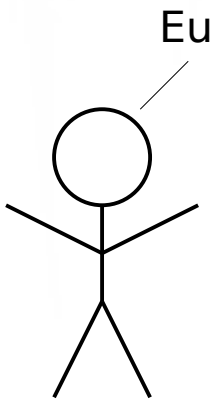


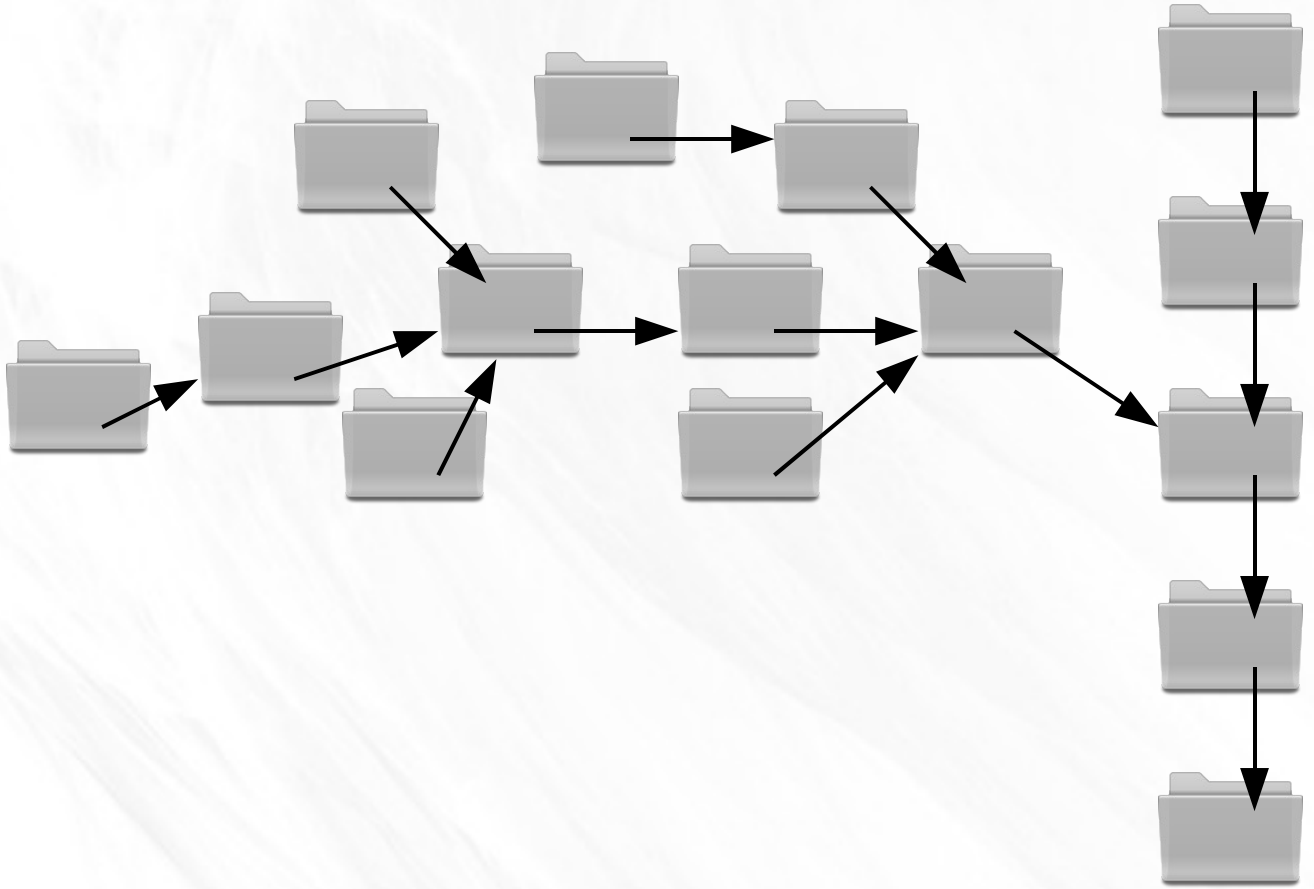
# Branches

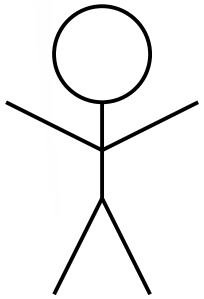
# Branches



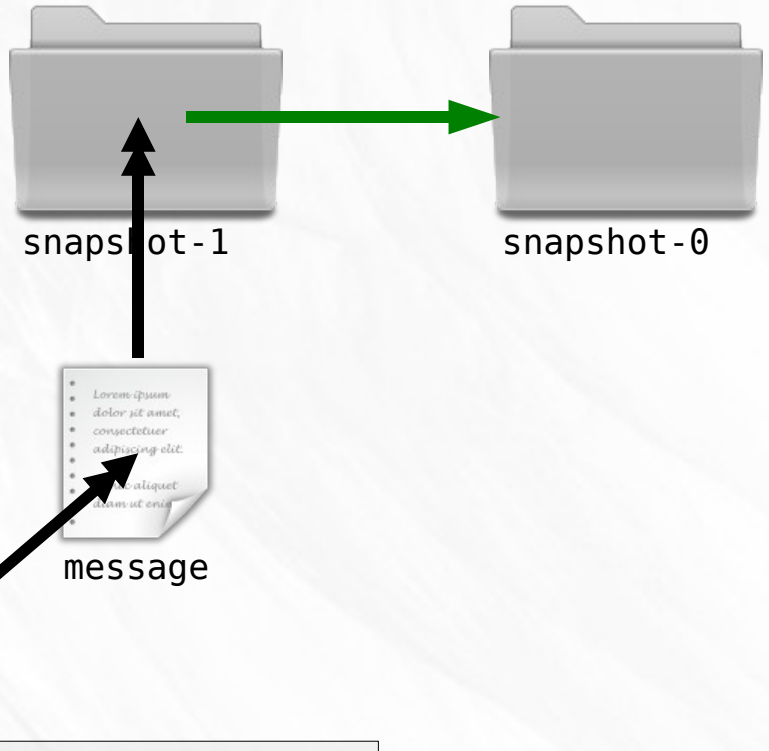


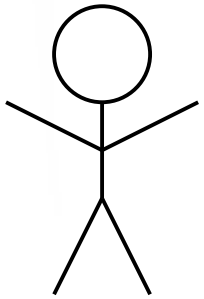
# Branches





# Branches

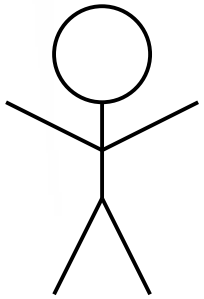


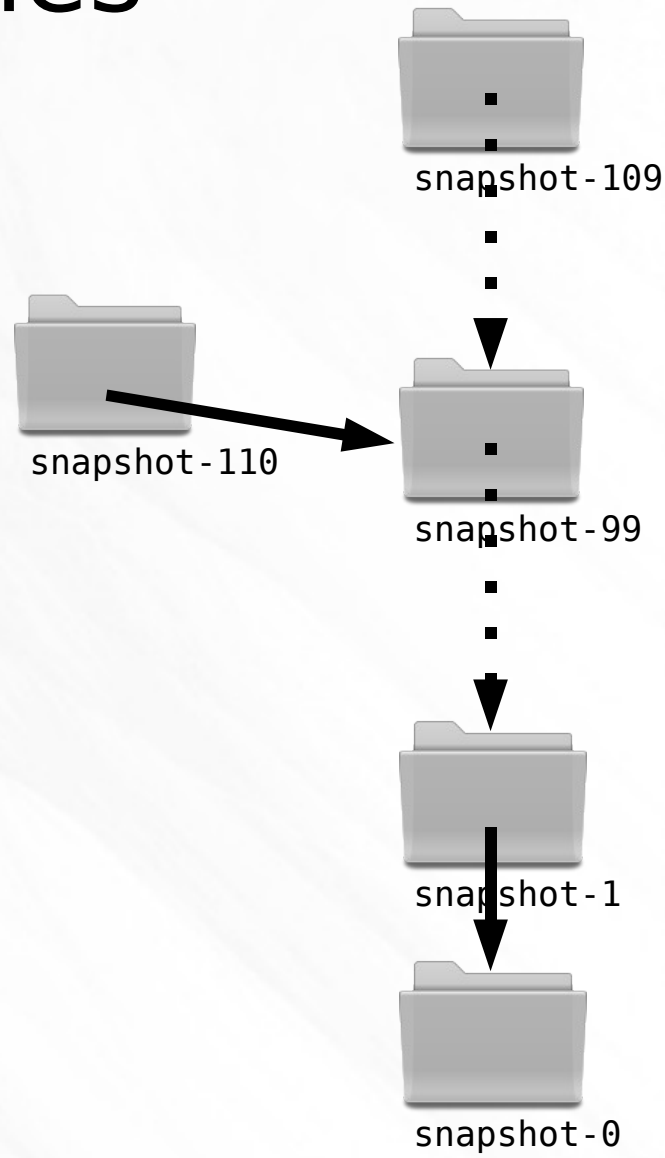



```
2009-05-21 23:45:01
parent: snapshot-0

Introduced a new foo,
and reset the bar to
xyzzy.
```

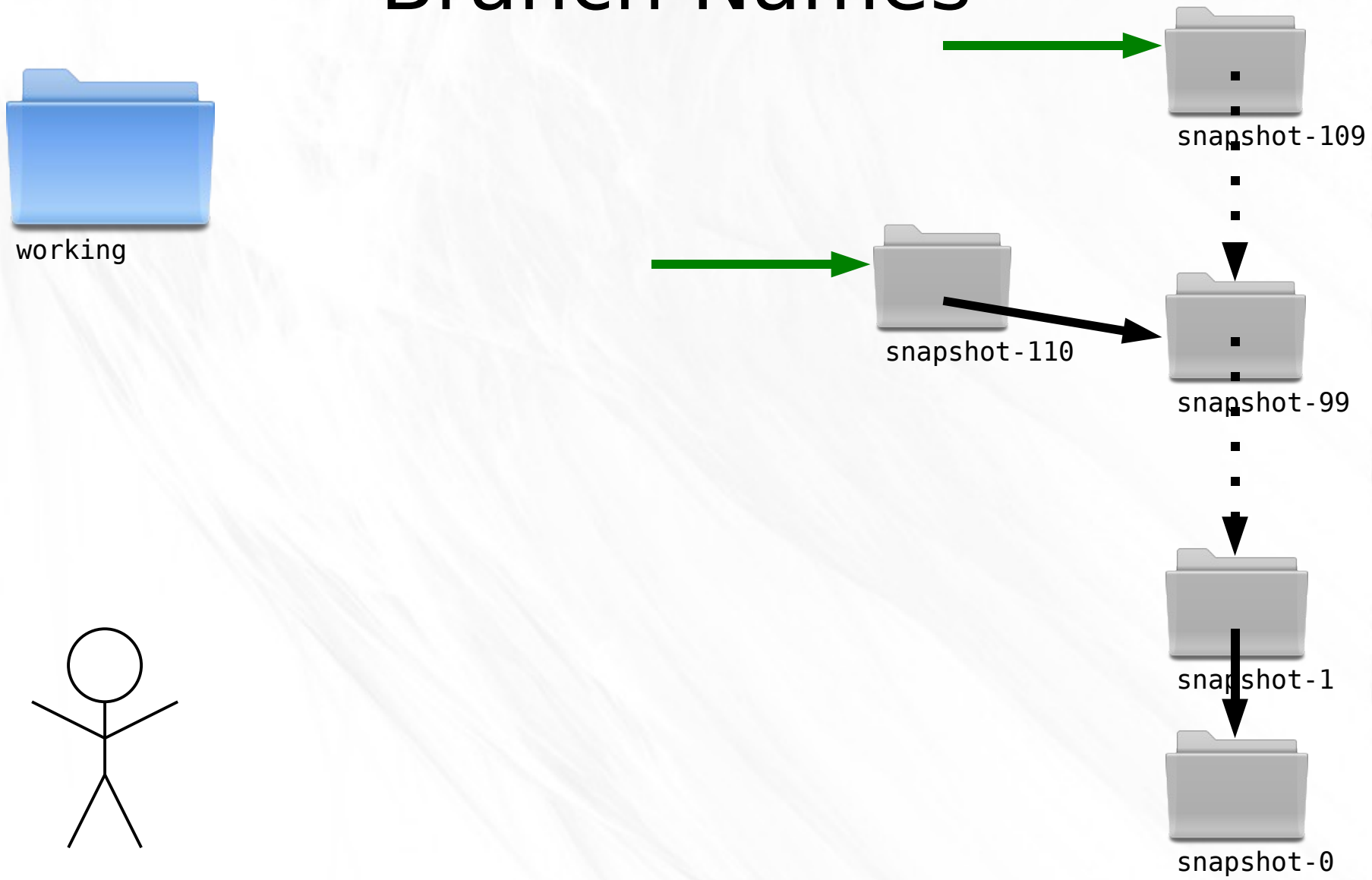
# Branch Names

working

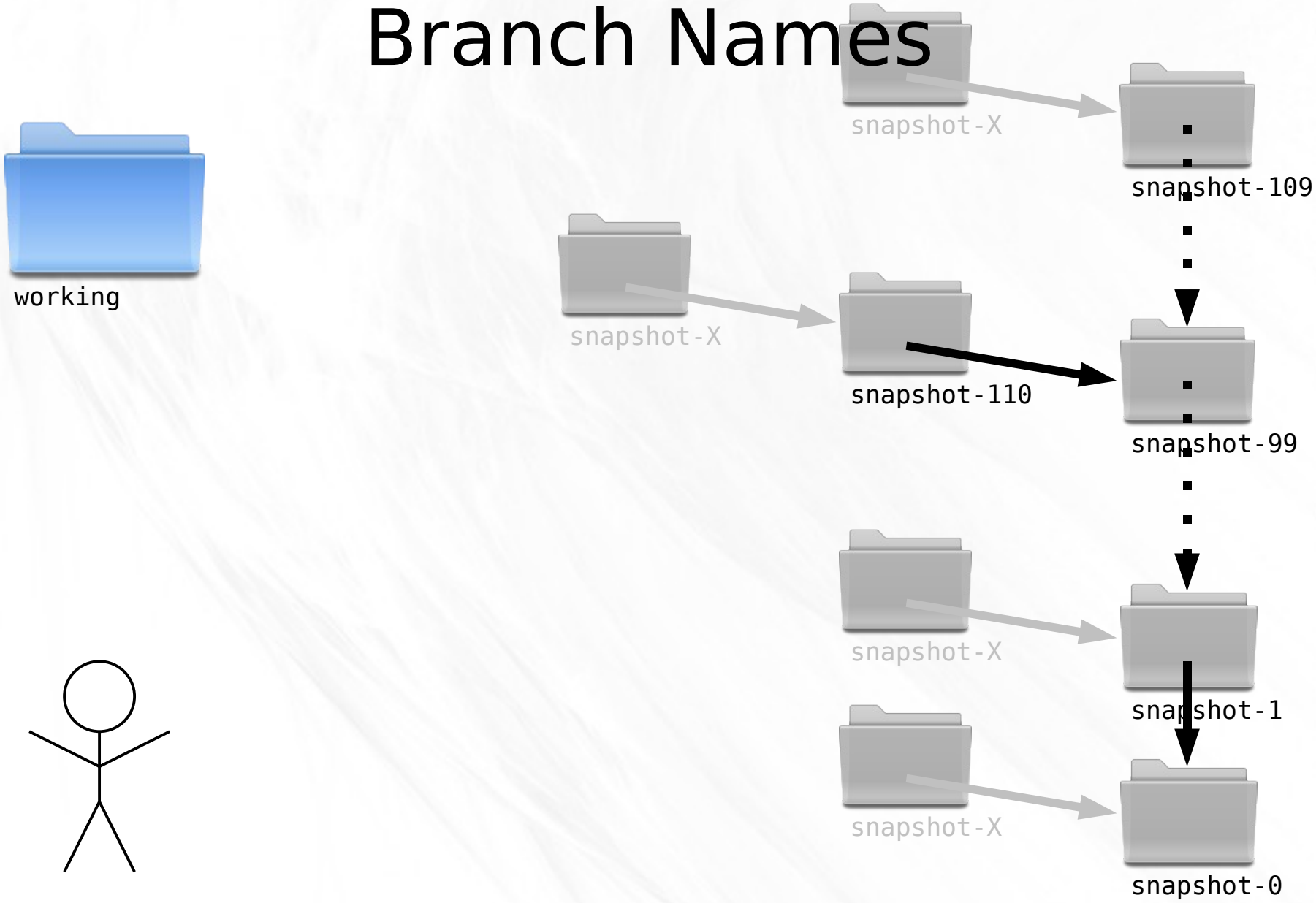




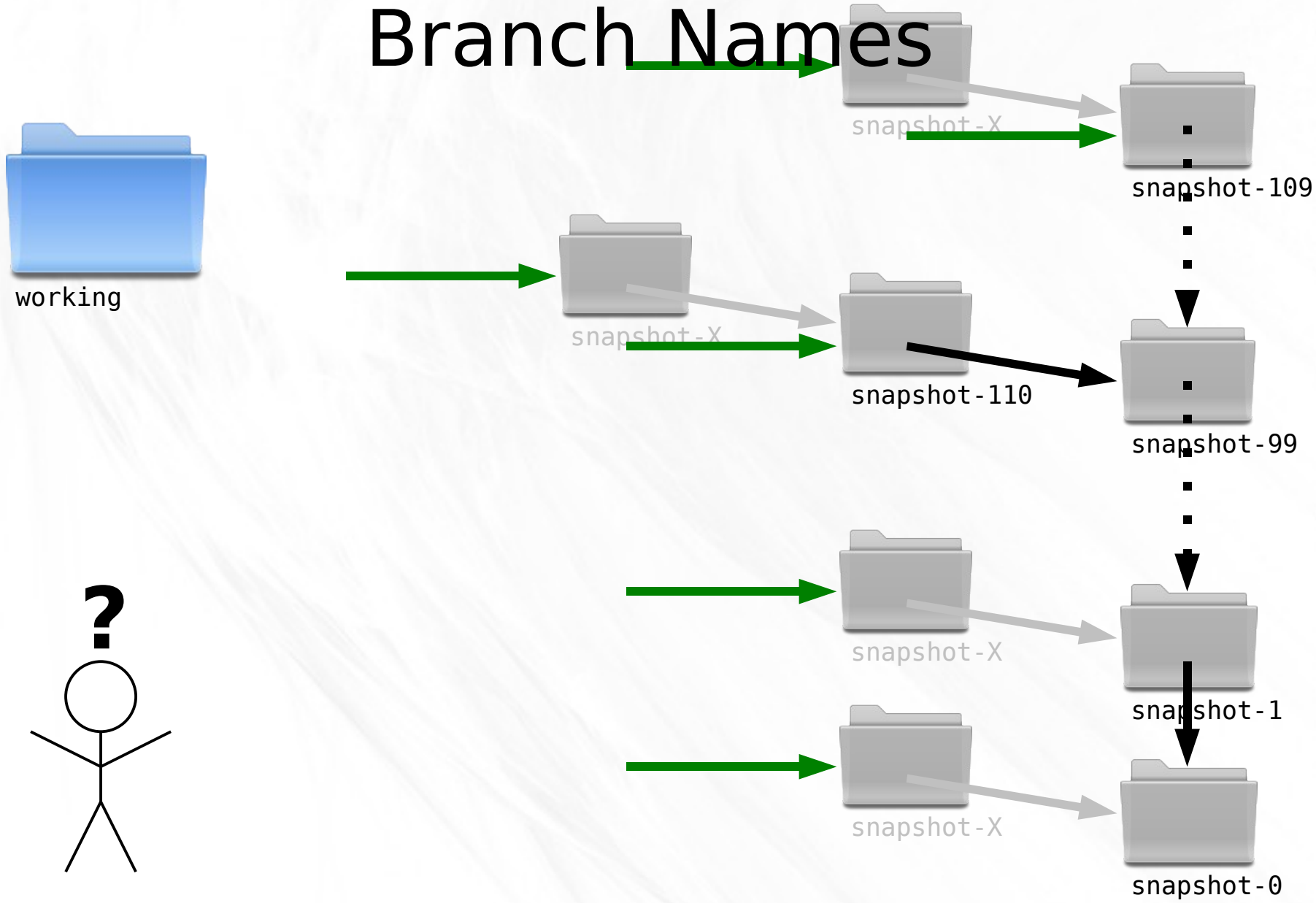
# Branch Names



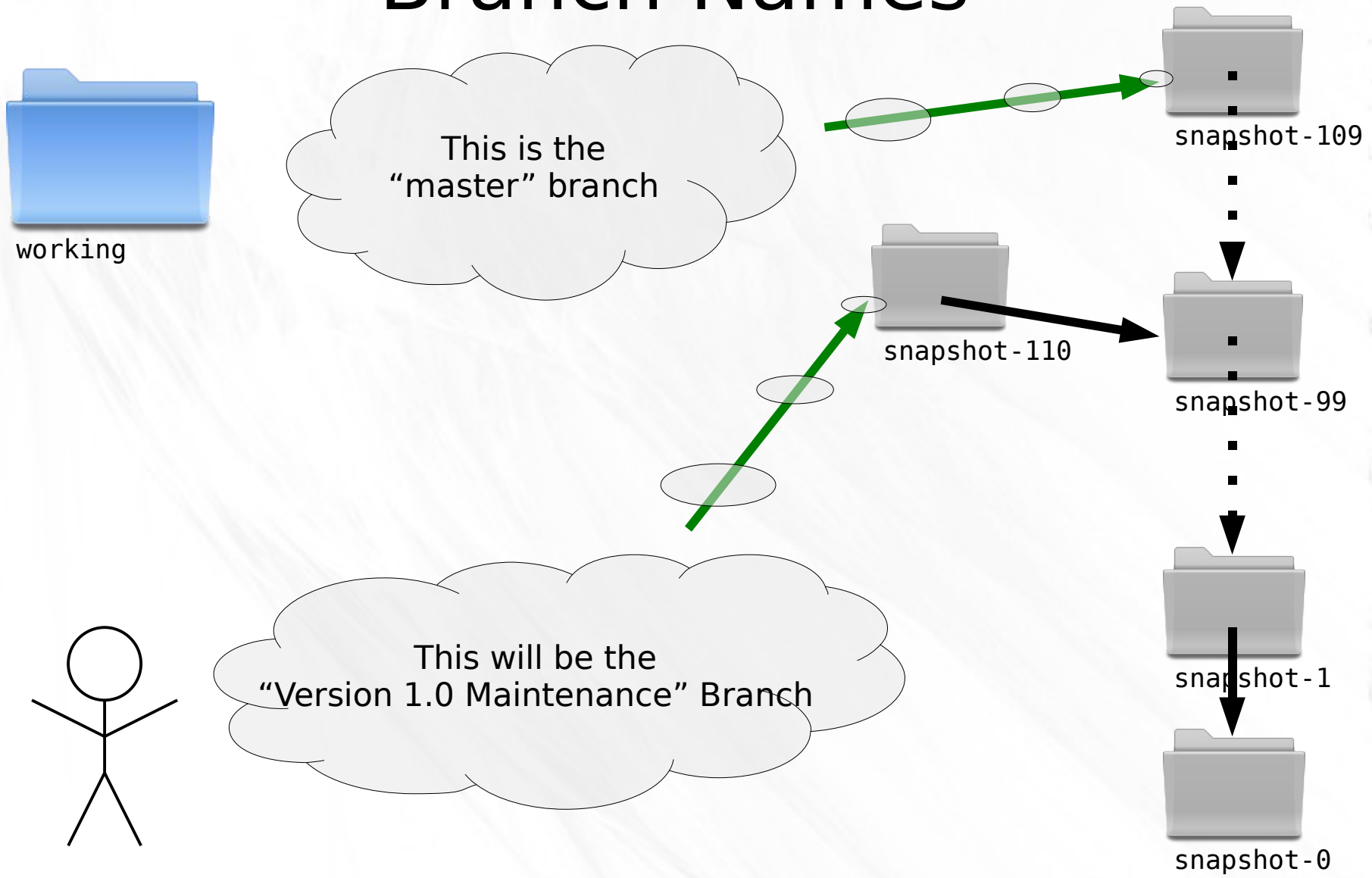
# Branch Names



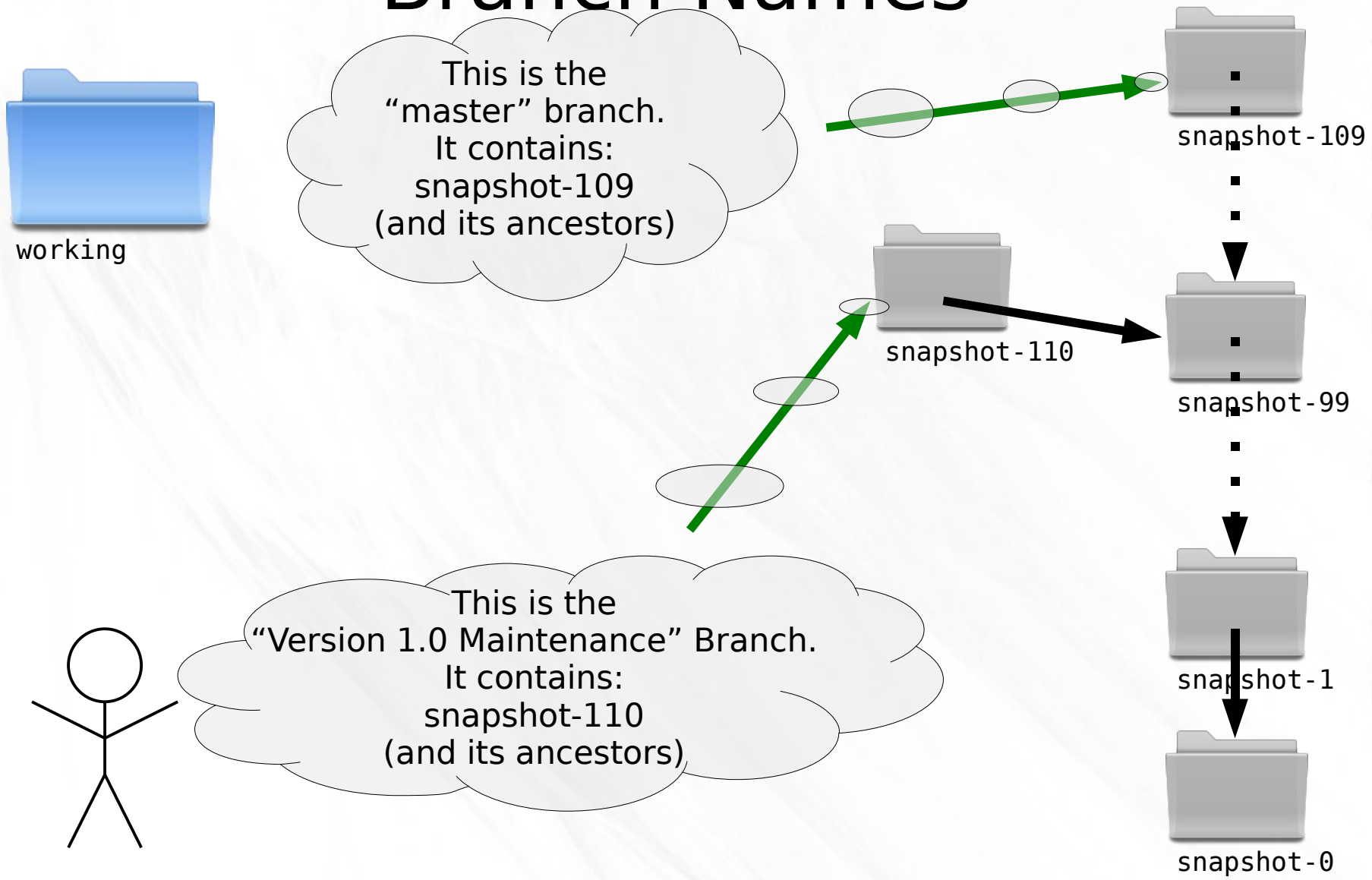
# Branch Names



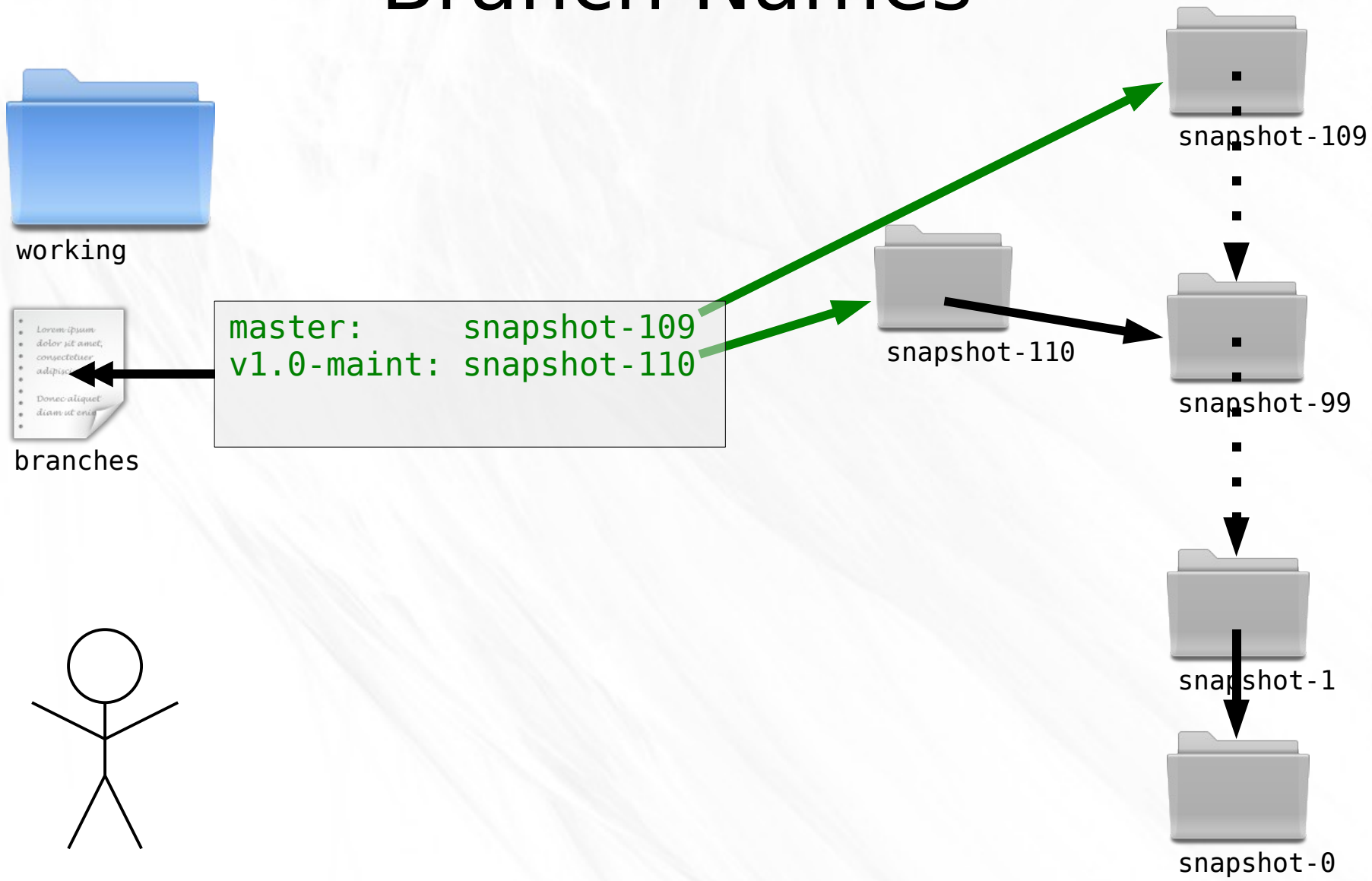
# Branch Names



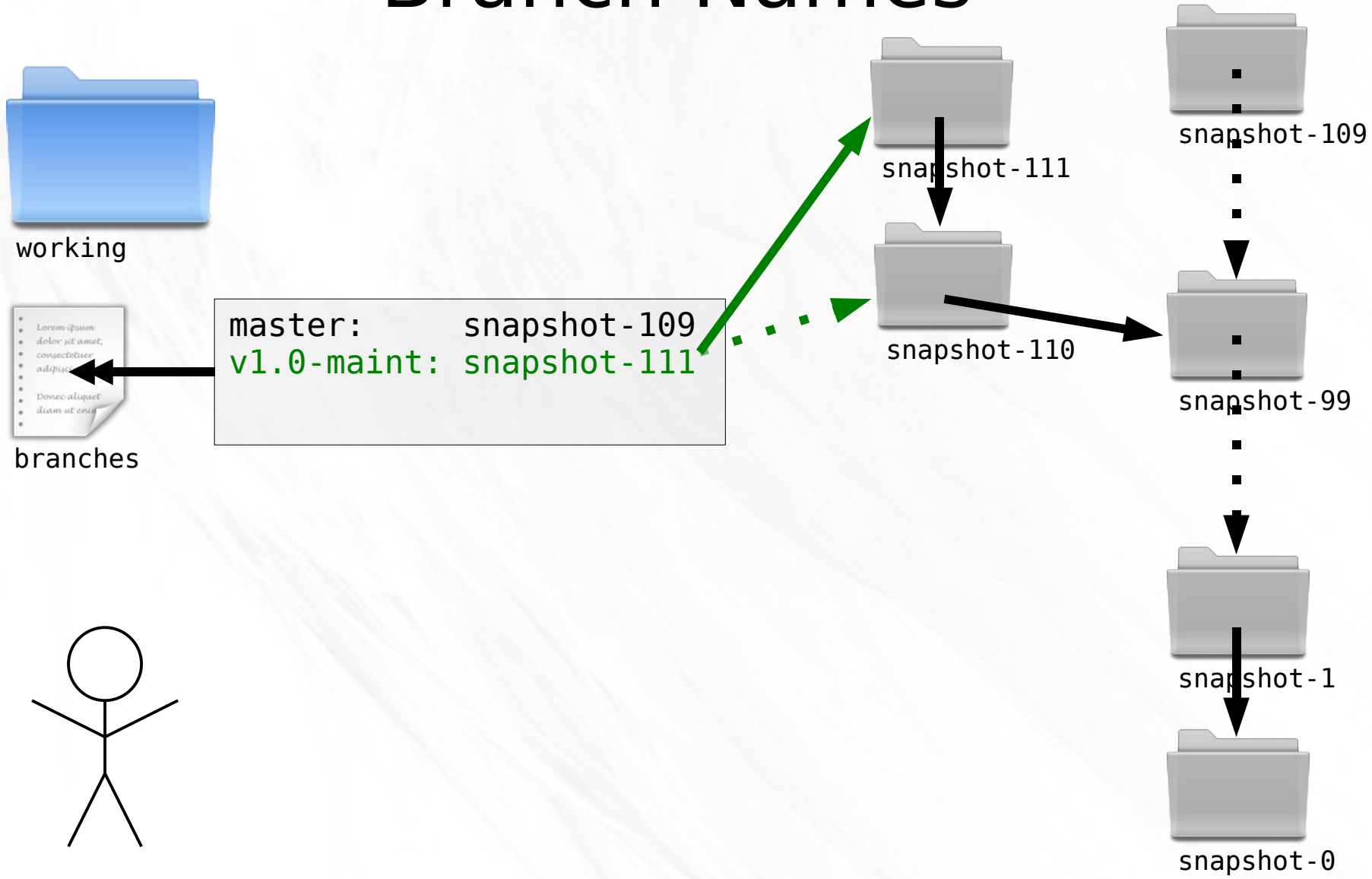
# Branch Names



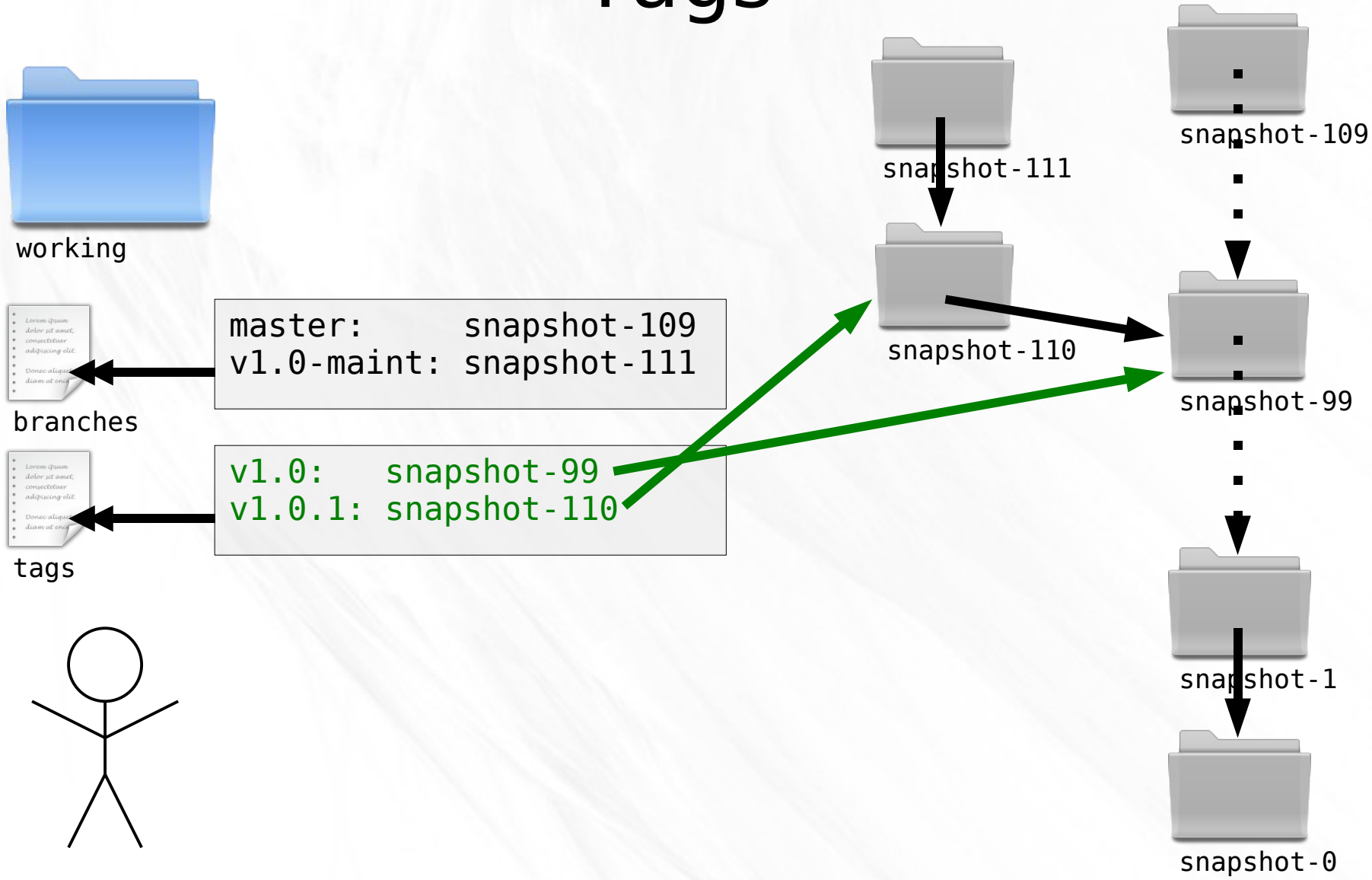
# Branch Names



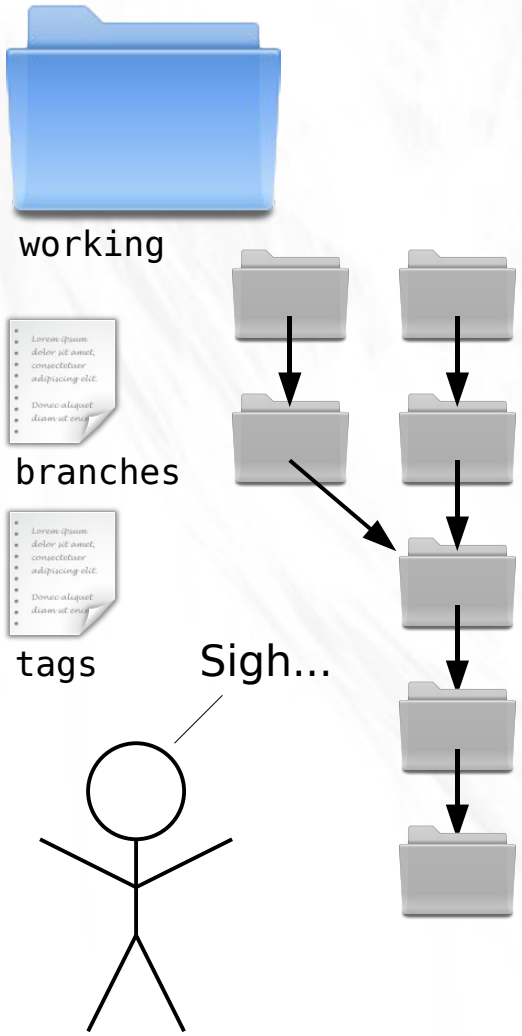
# Branch Names



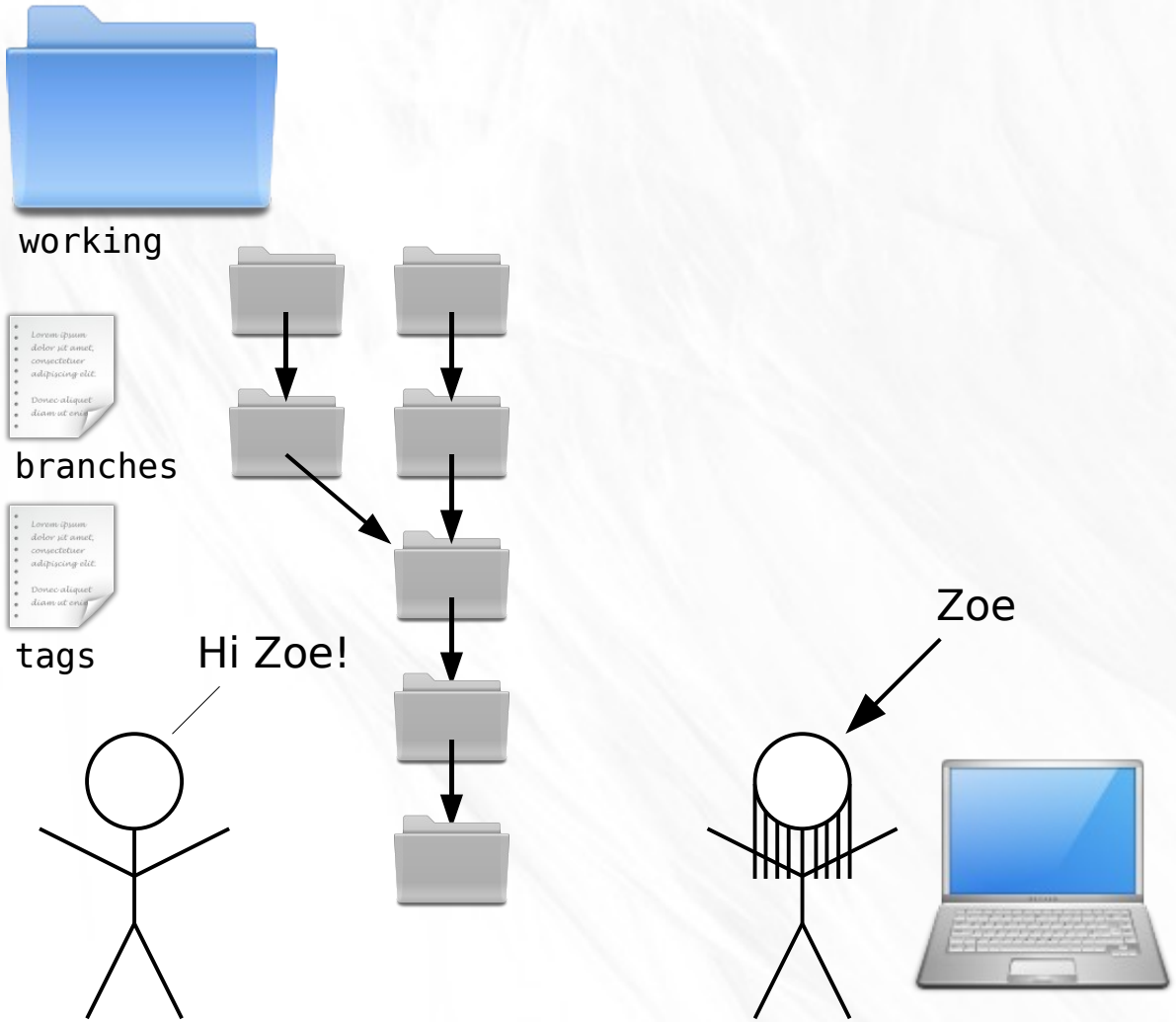
# Tags


working
branches
master: snapshot-109
v1.0-maint: snapshot-111
tags
v1.0: snapshot-99
v1.0.1: snapshot-110
snapshot-111
snapshot-109
snapshot-110
snapshot-99
snapshot-1
snapshot-0

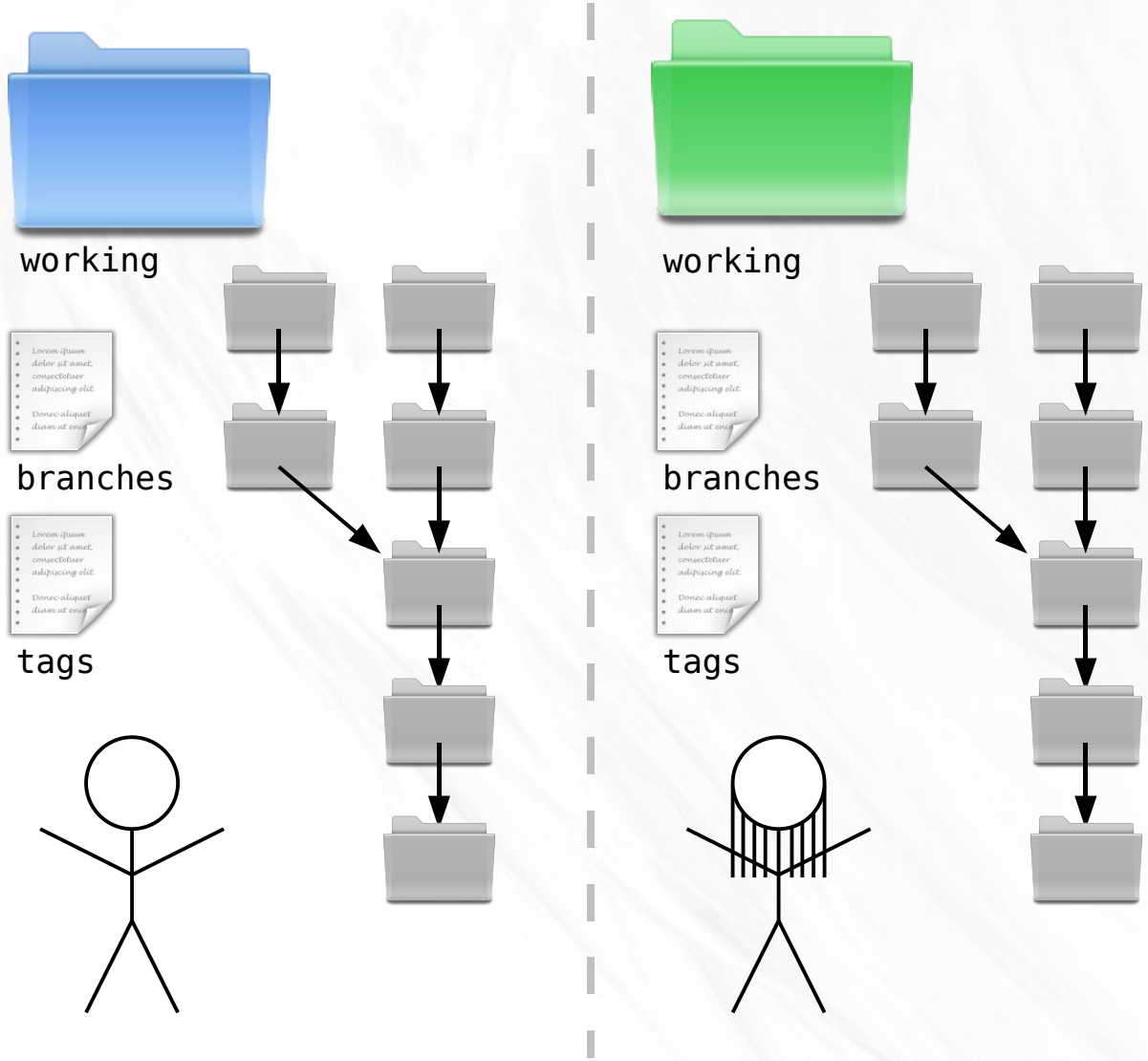
# Distributed



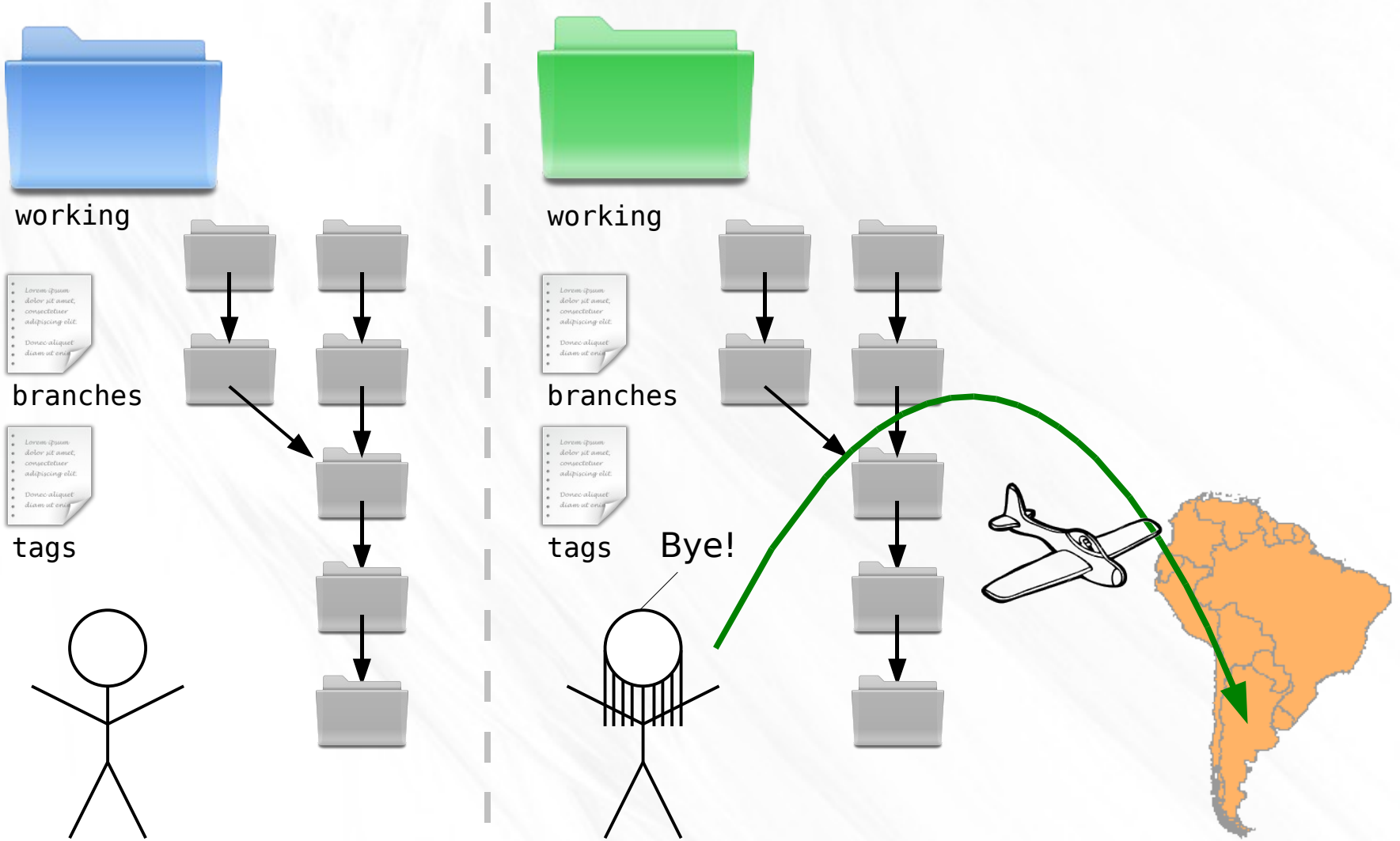
# Distributed



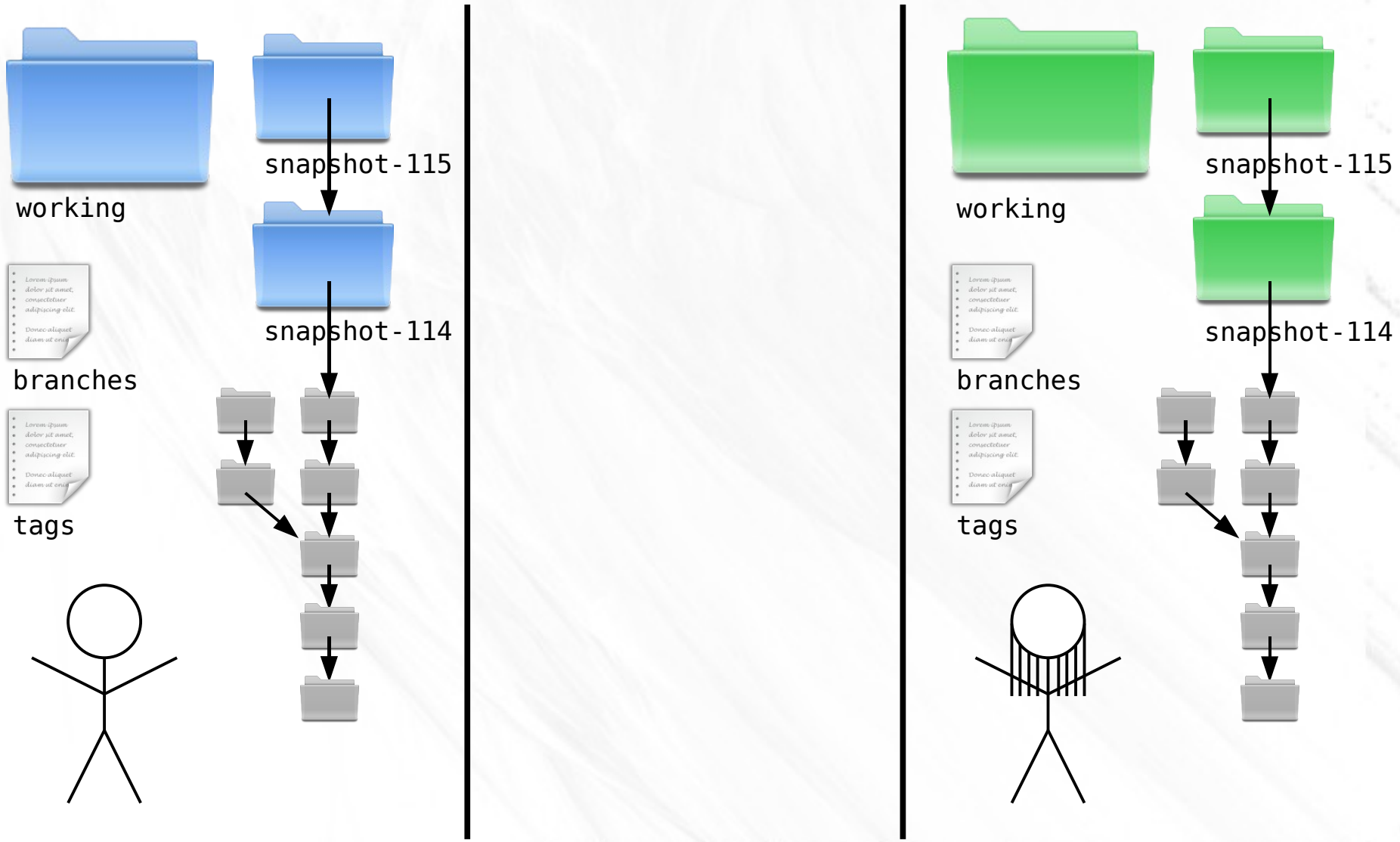
# Distributed



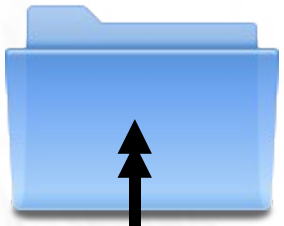
# Distributed



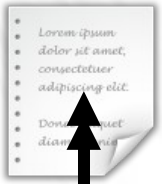
# Distributed



# Distributed

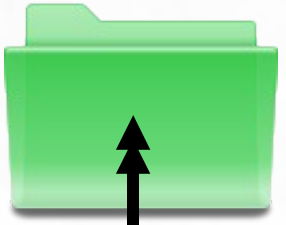


snapshot-114



message

```
2009-05-22 12:12:12
parent: snapshot-113
author: Me <me@me.me>

Blarfle, a cool new
feature; extends the
existing blorg.
```



snapshot-114

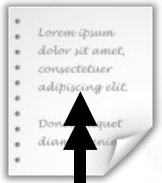
message

```
2009-05-21 23:45:01
parent: snapshot-113
author: Zoe <zoe@z.oe>

Introduced a new foo,
and reset the bar to
xyzzy.
```

# Distributed

8ba3441b6b89cad23387ee875f2ae55069291f4b

**SHA1**

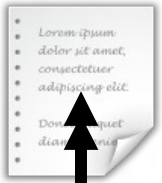


message

```
2009-05-22 12:12:12
parent: snapshot-113
author: Me <me@me.me>

Blarfle, a cool new
feature; extends the
existing blorg.
```

db9ecb5b5a6294a8733503ab57577db96ff2249e

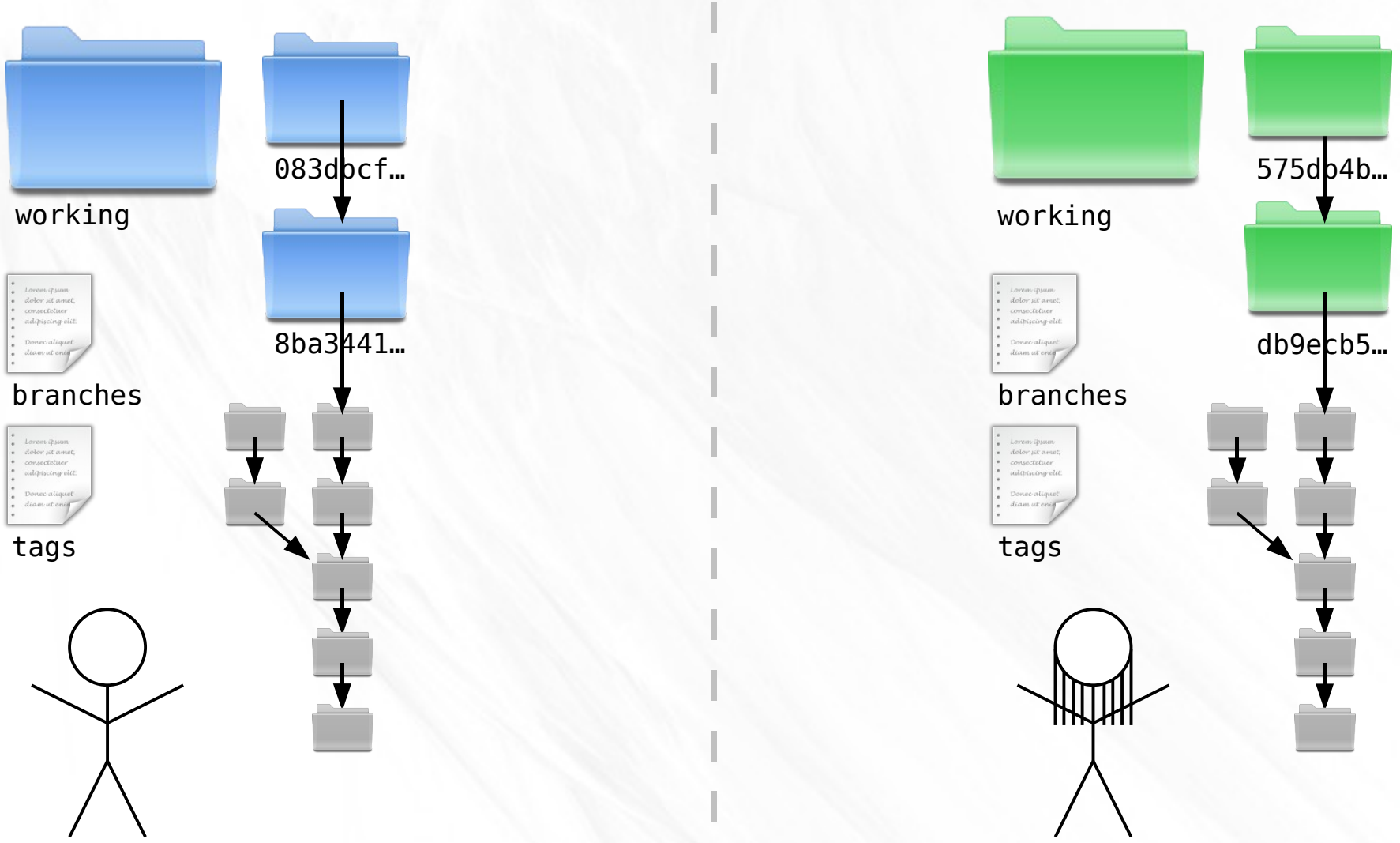
**SHA1**



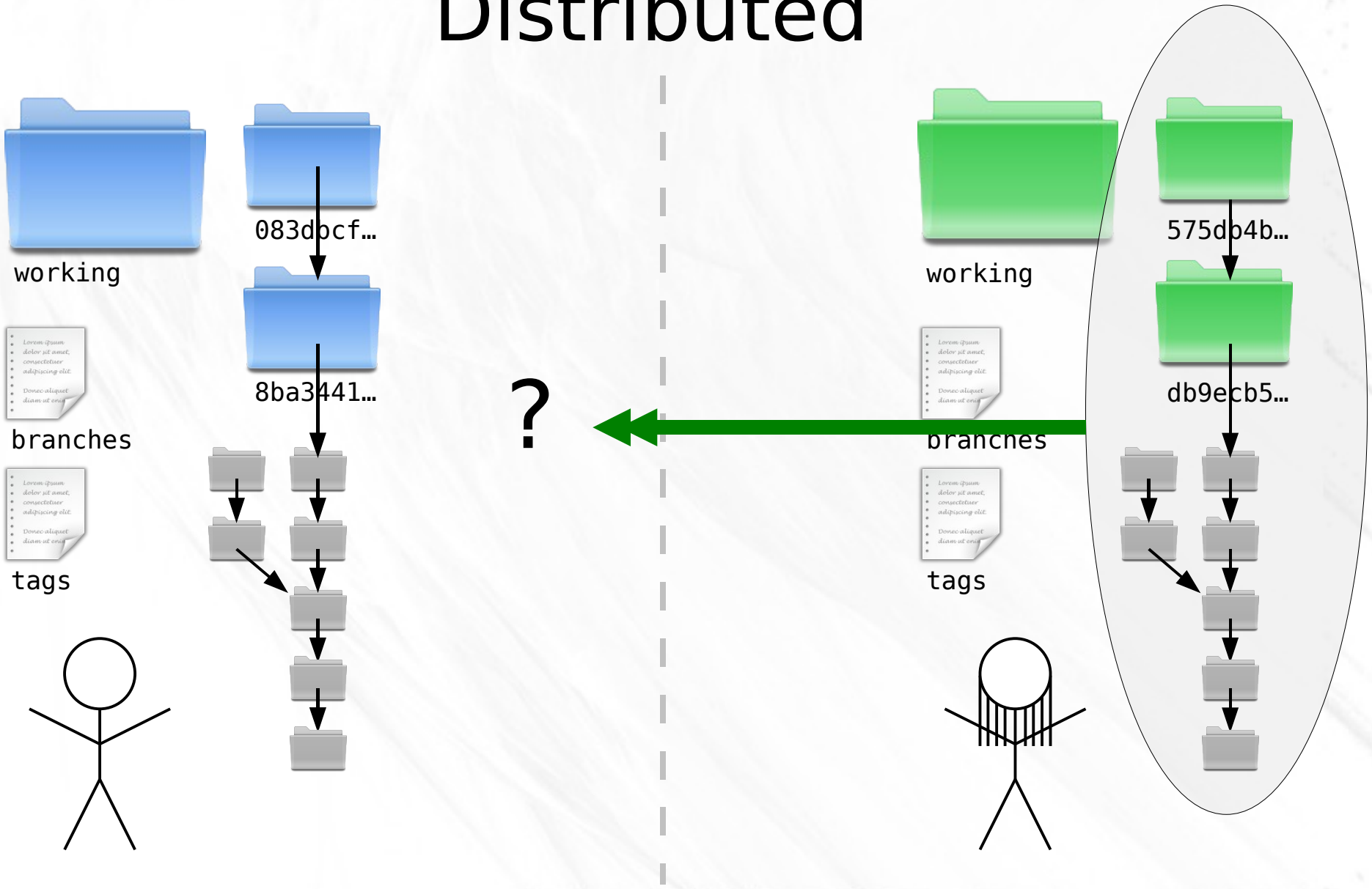
message

```
2009-05-21 23:45:01
parent: snapshot-113
author: Zoe <zoe@z.oe>

Introduced a new foo,
and reset the bar to
xyzzy.
```

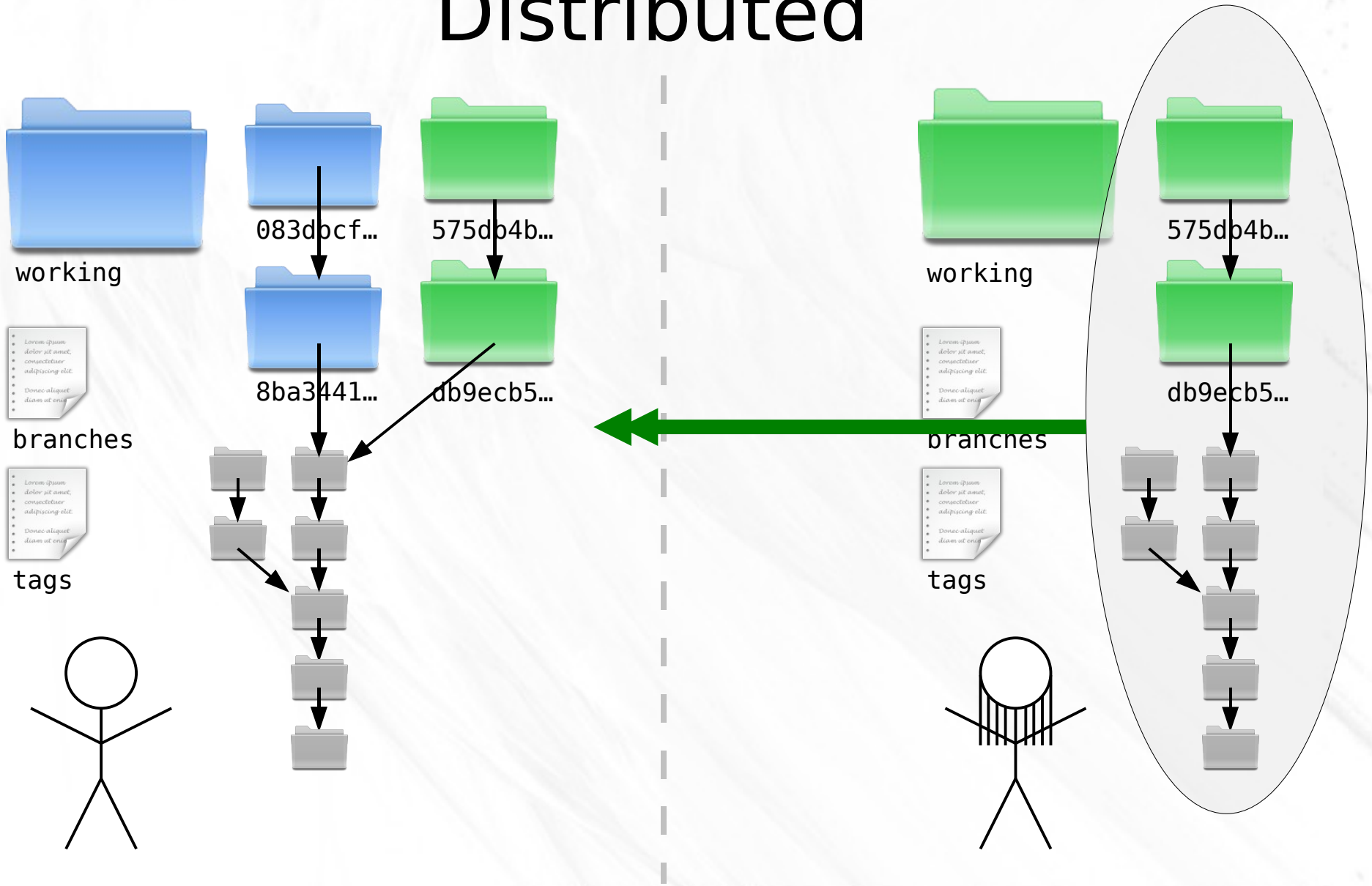
# Distributed



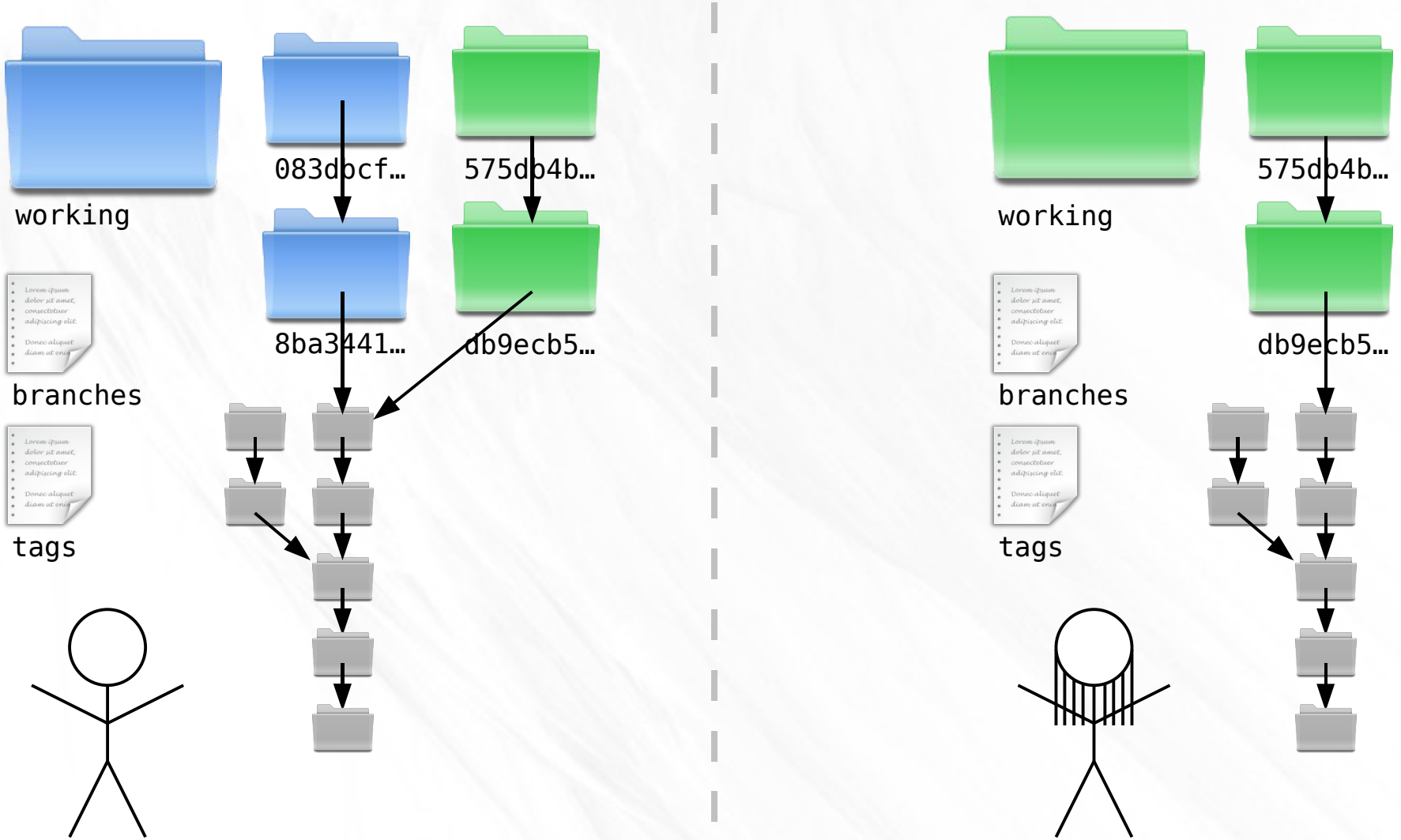
# Distributed



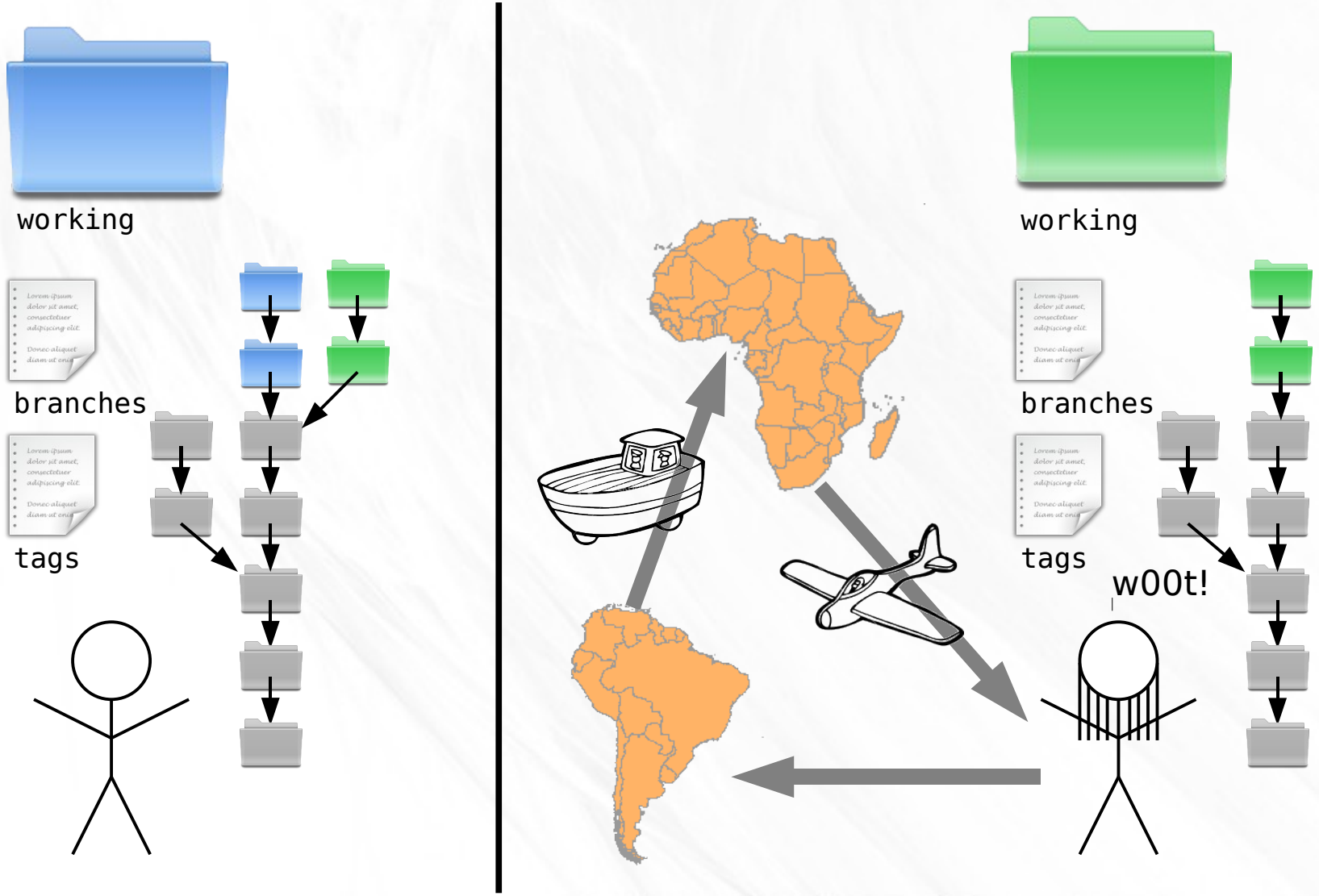
# Distributed



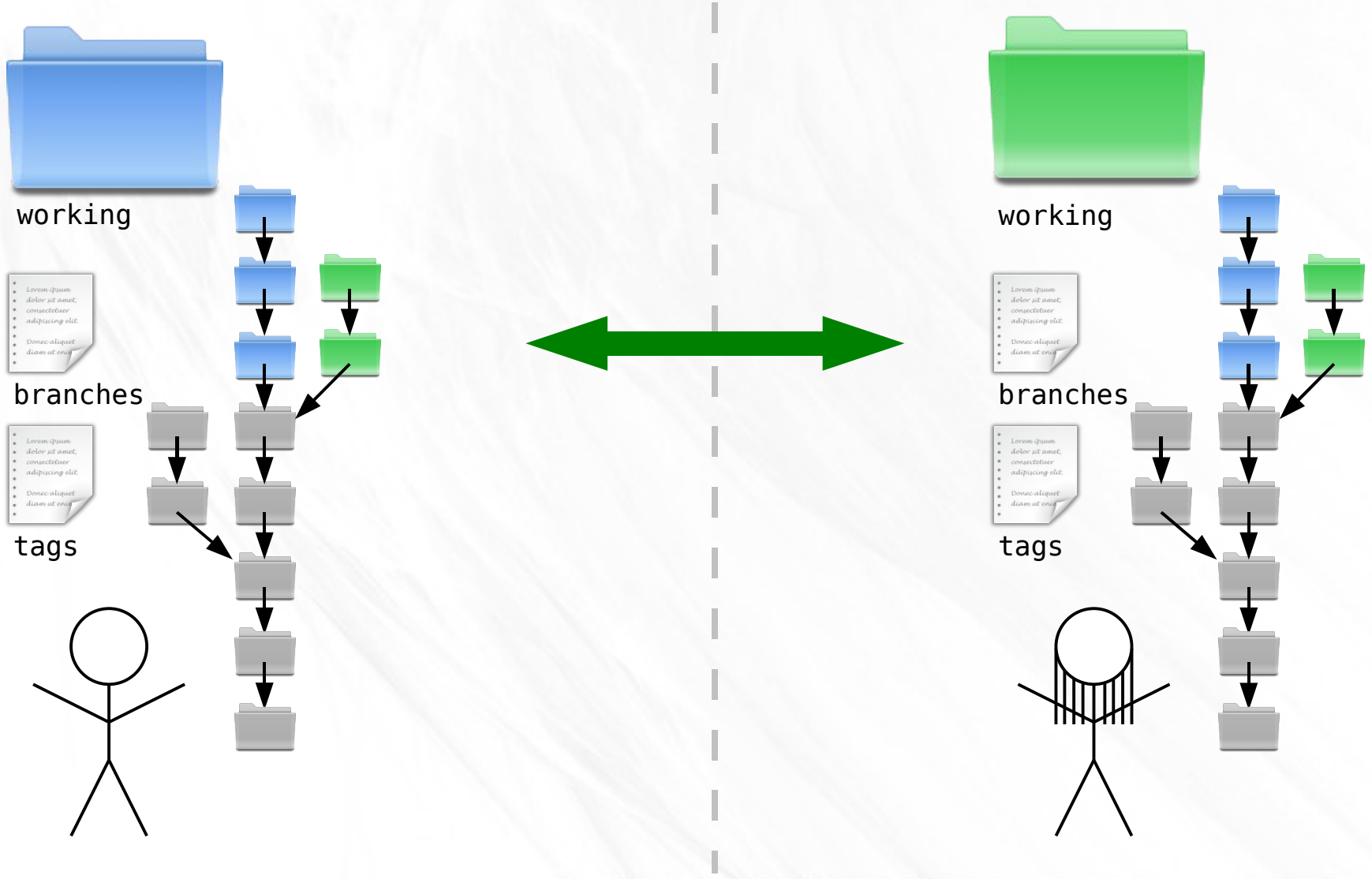
# Distributed



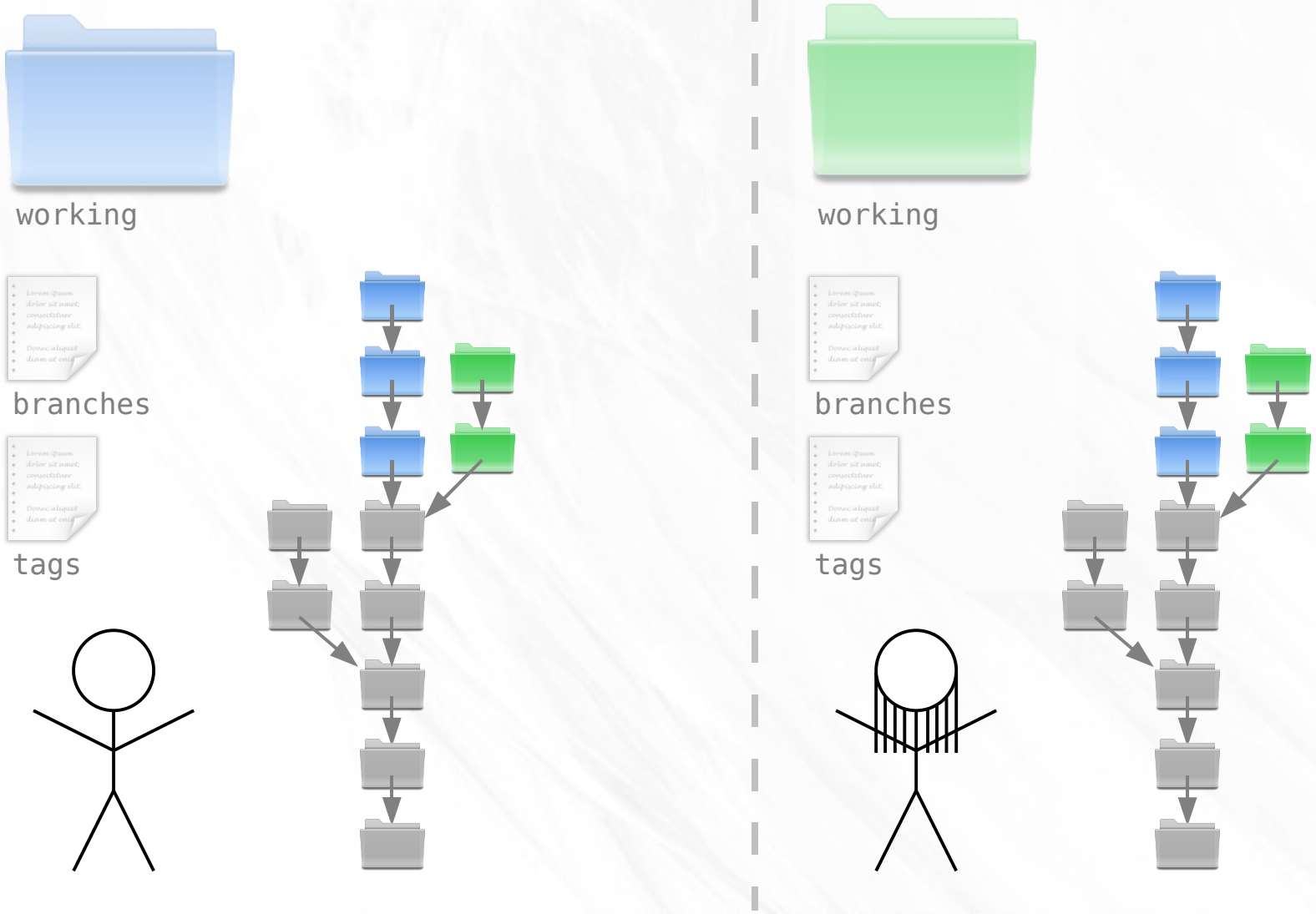
# Offline



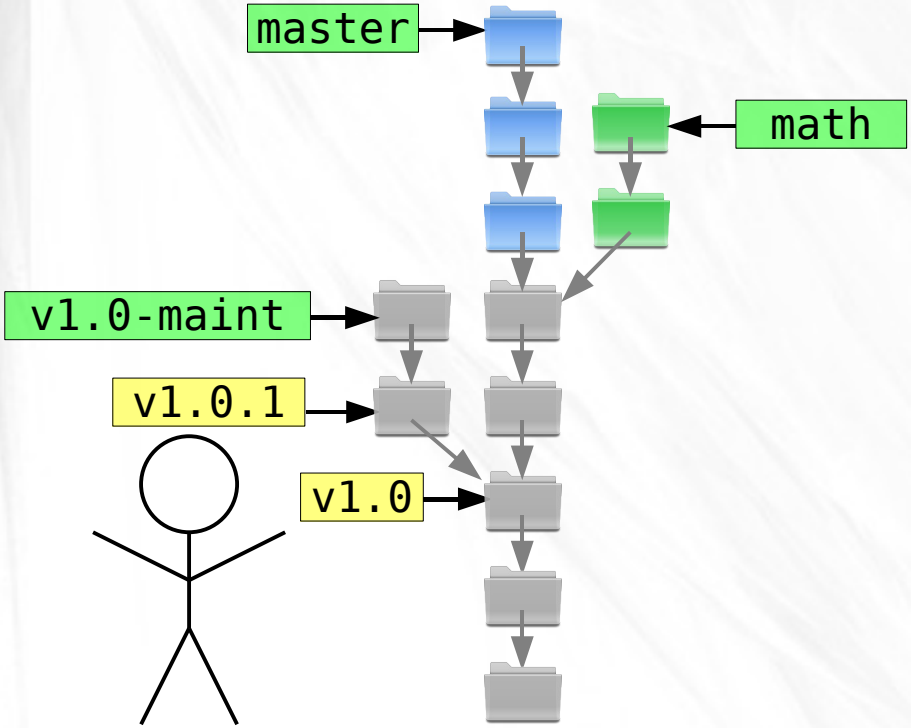
# Offline

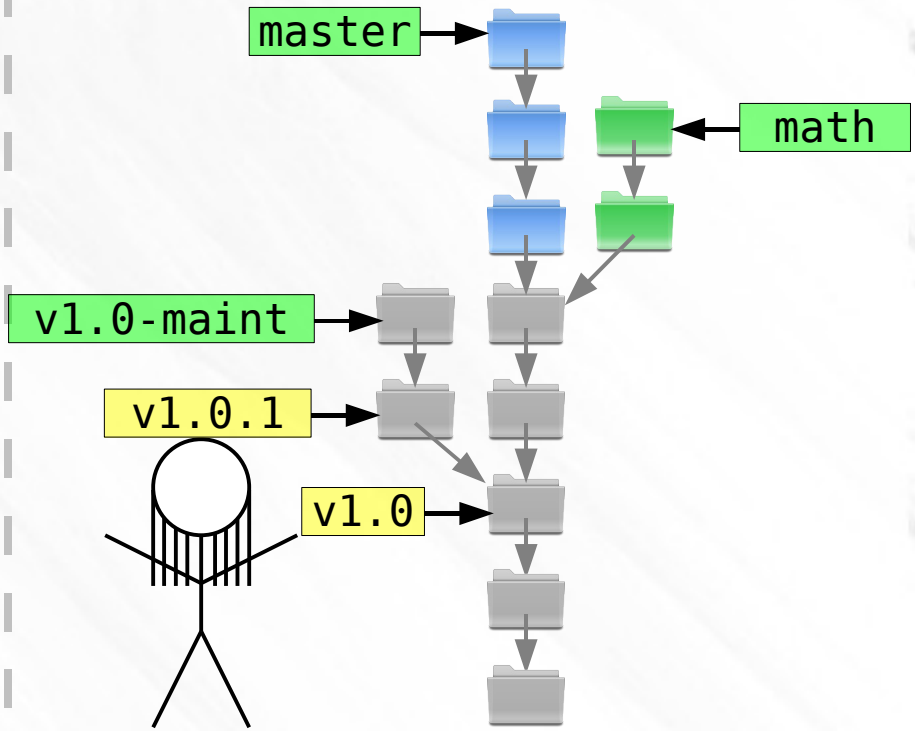


# (simpler drawings)

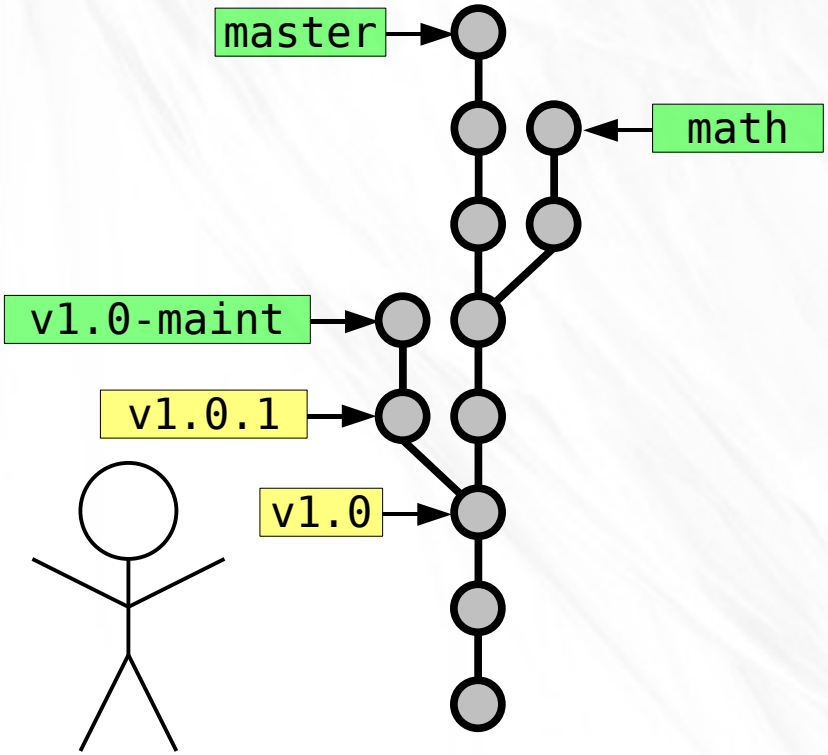


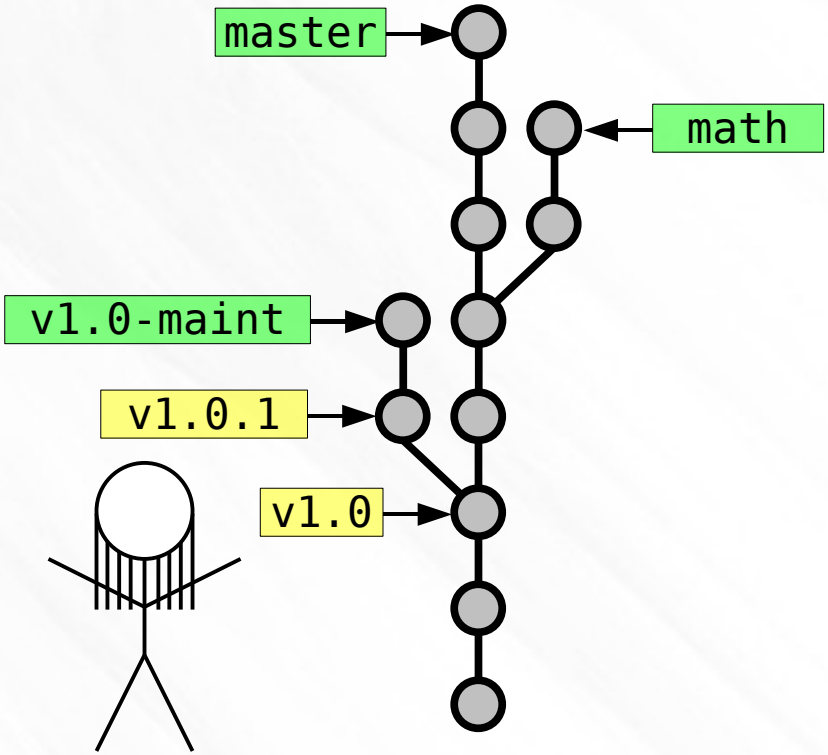
# (simpler drawings)



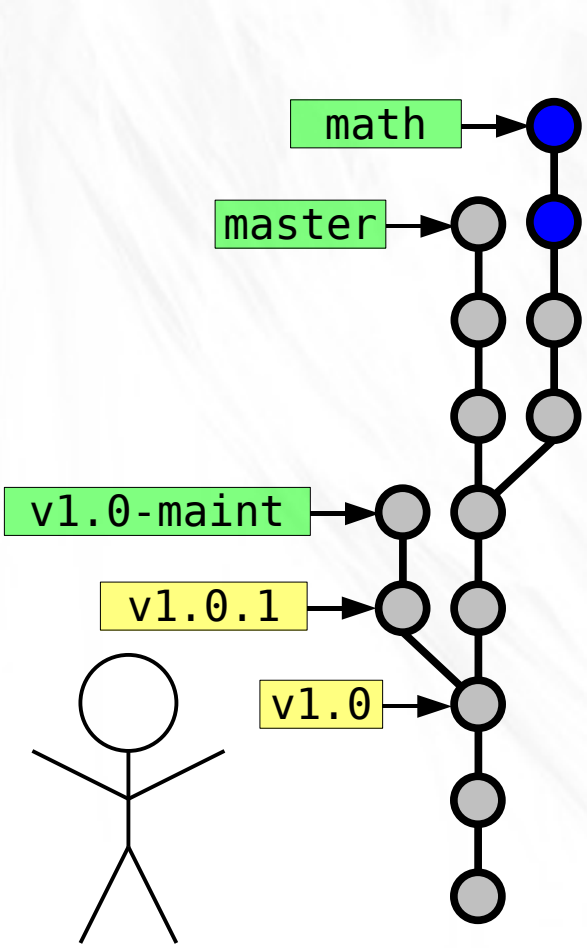



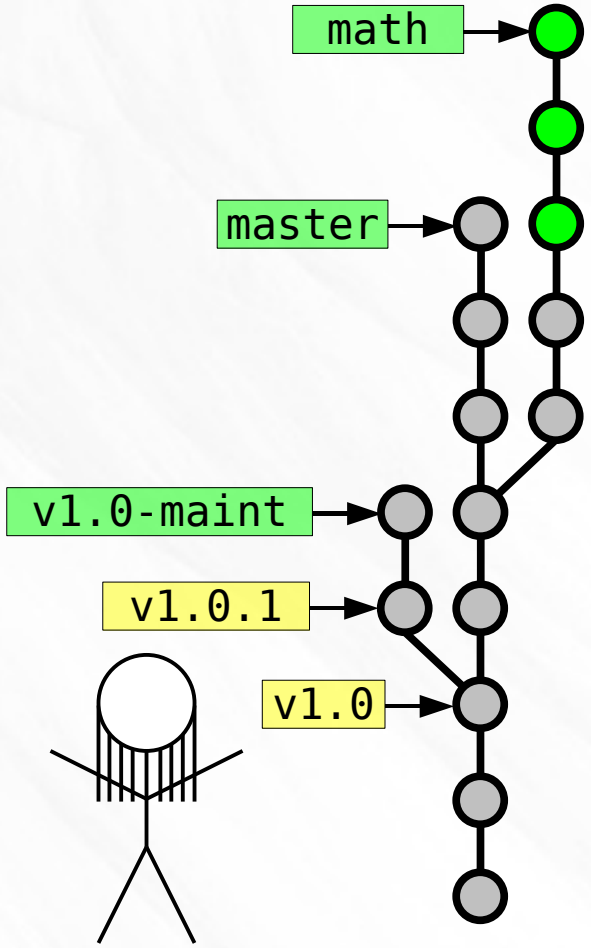
# (simpler drawings)



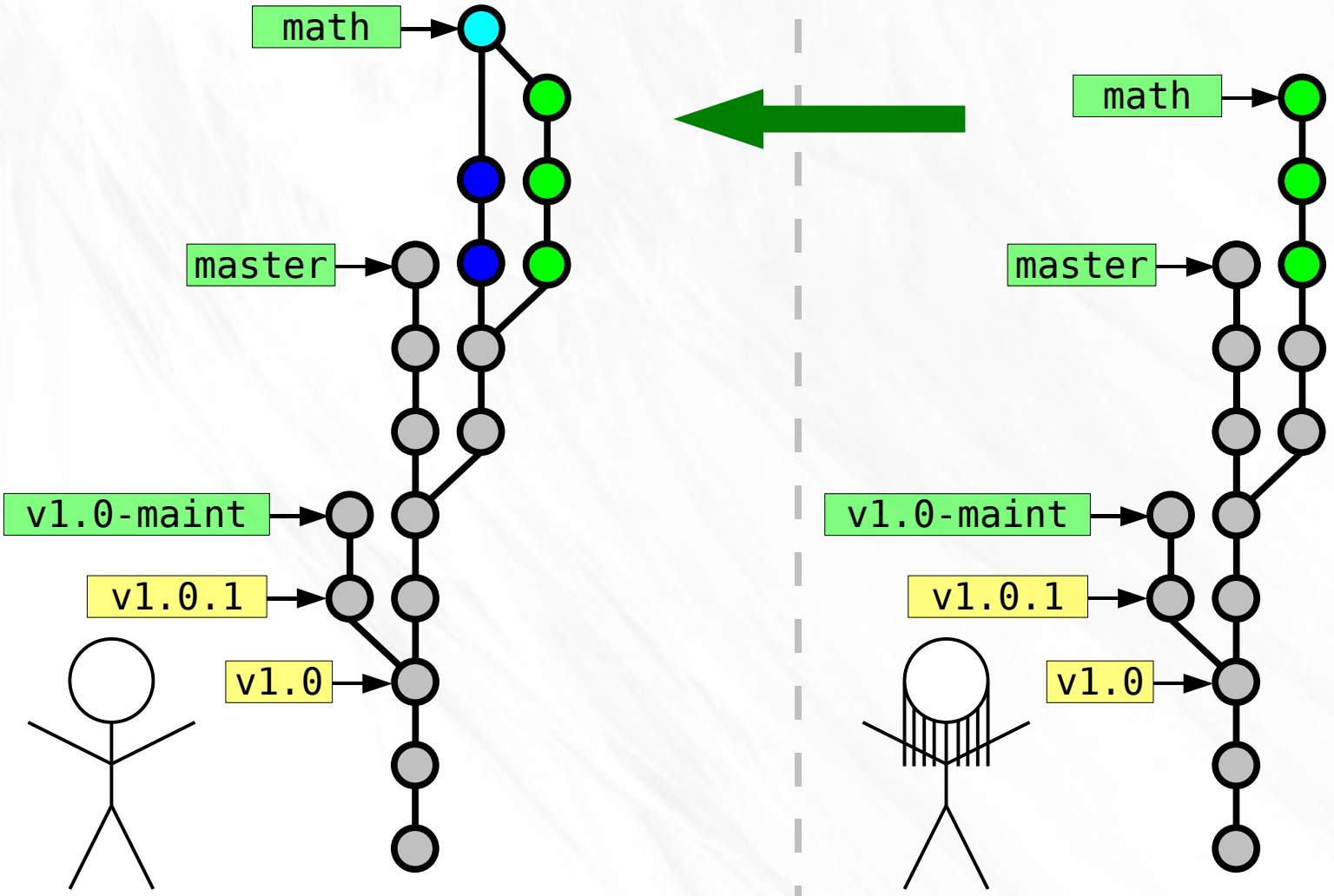



# Merges

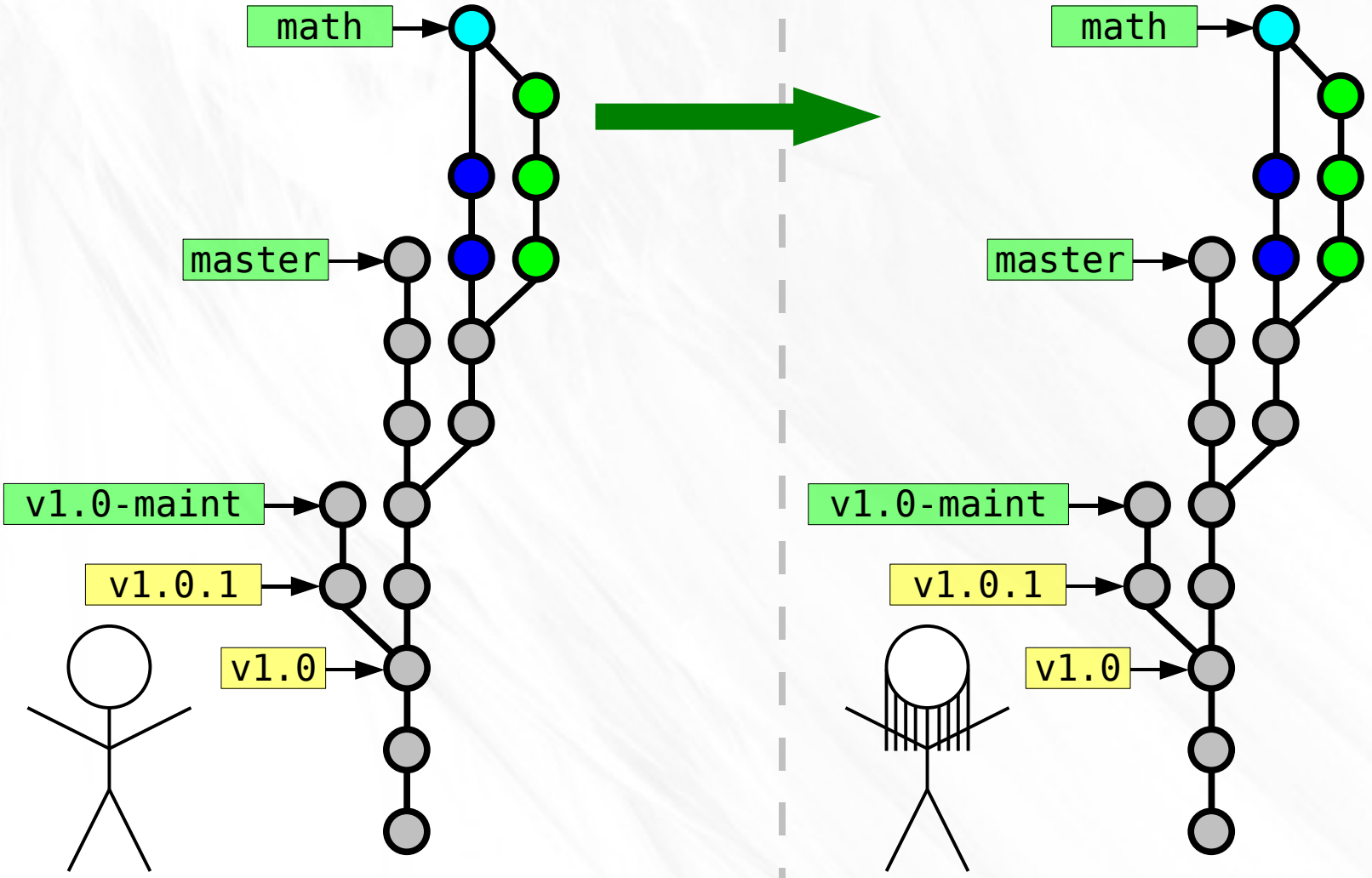

math
master
v1.0-maint
v1.0.1
v1.0


math
master
v1.0-maint
v1.0.1
v1.0

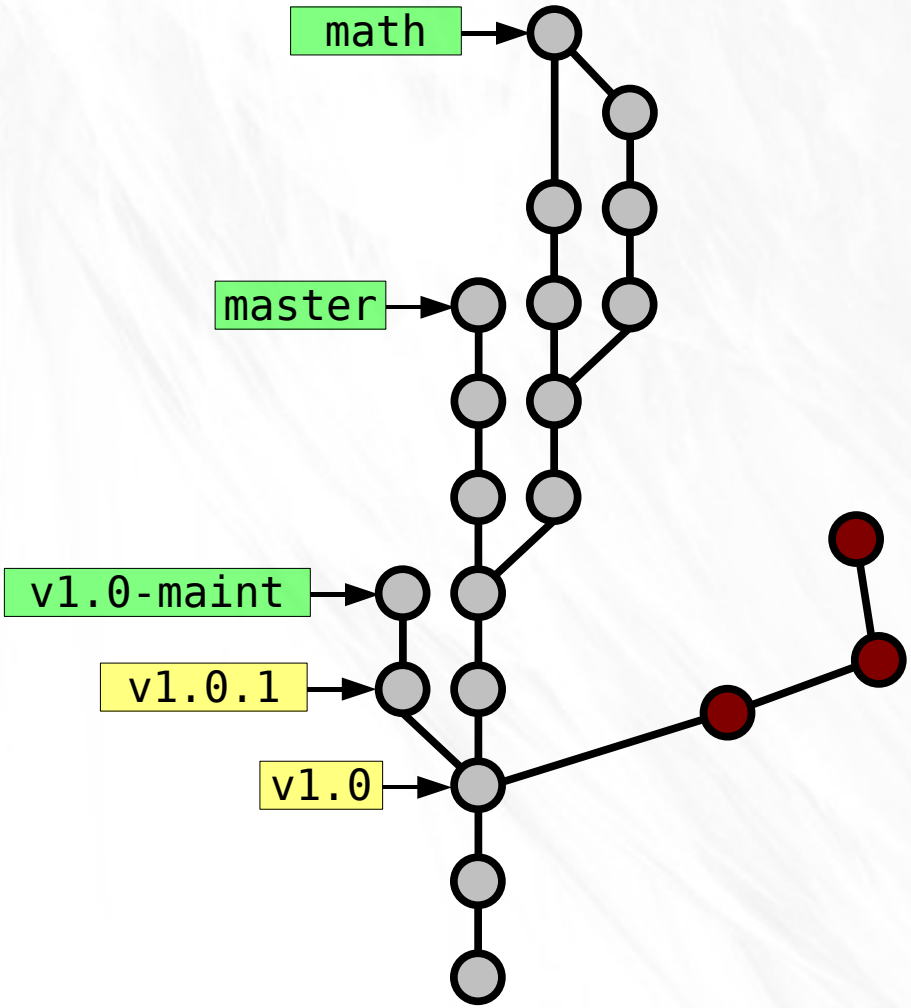
# Merges

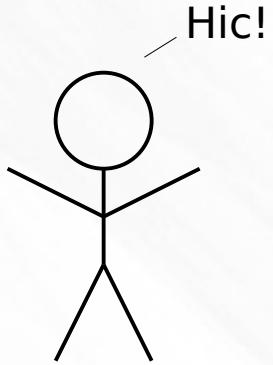

math
master
v1.0-maint
v1.0.1
v1.0
math
master
v1.0-maint
v1.0.1
v1.0

# Merges

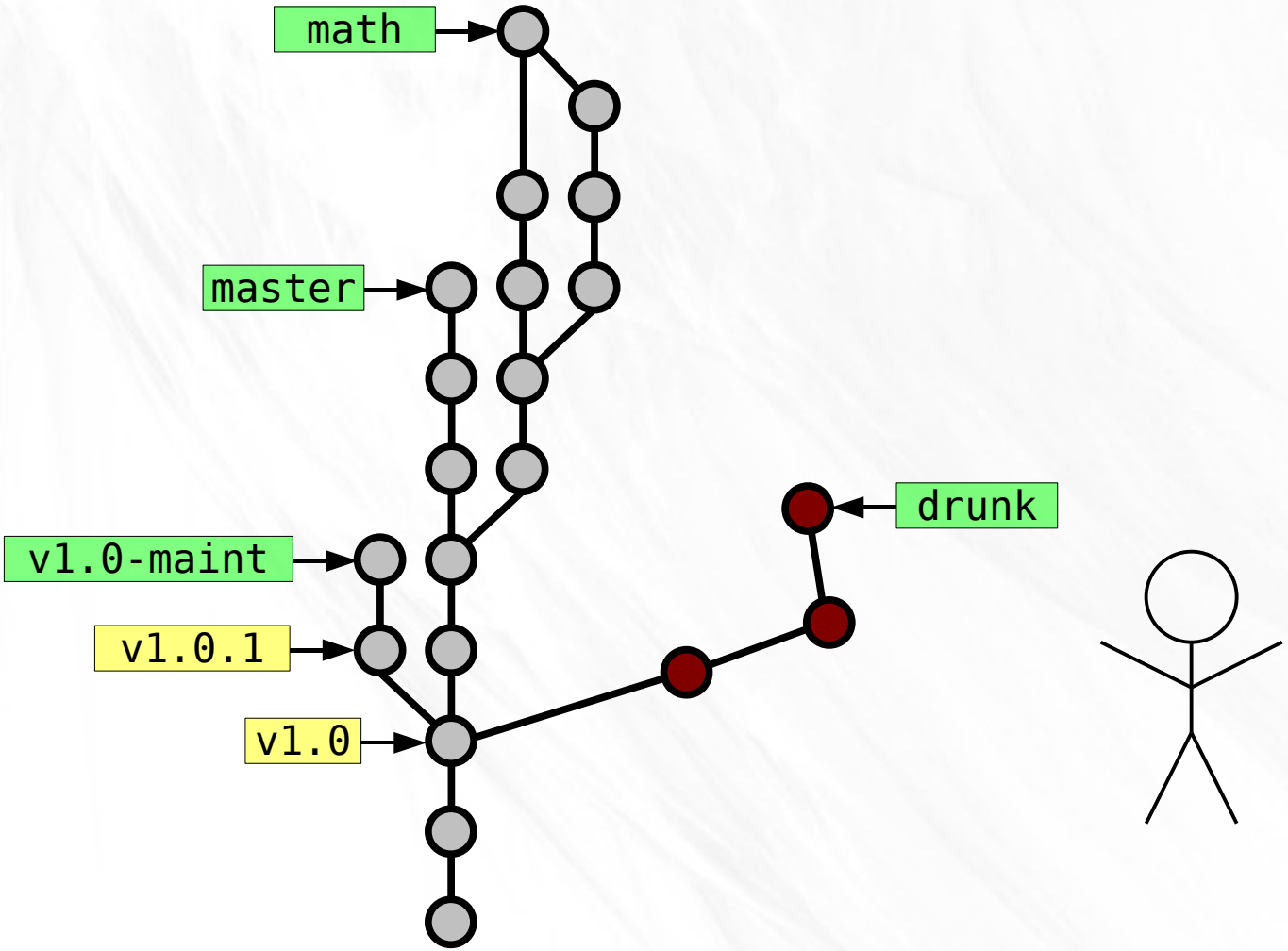

math
master
v1.0-maint
v1.0.1
v1.0
math
master
v1.0-maint
v1.0.1
v1.0

# Rewriting History

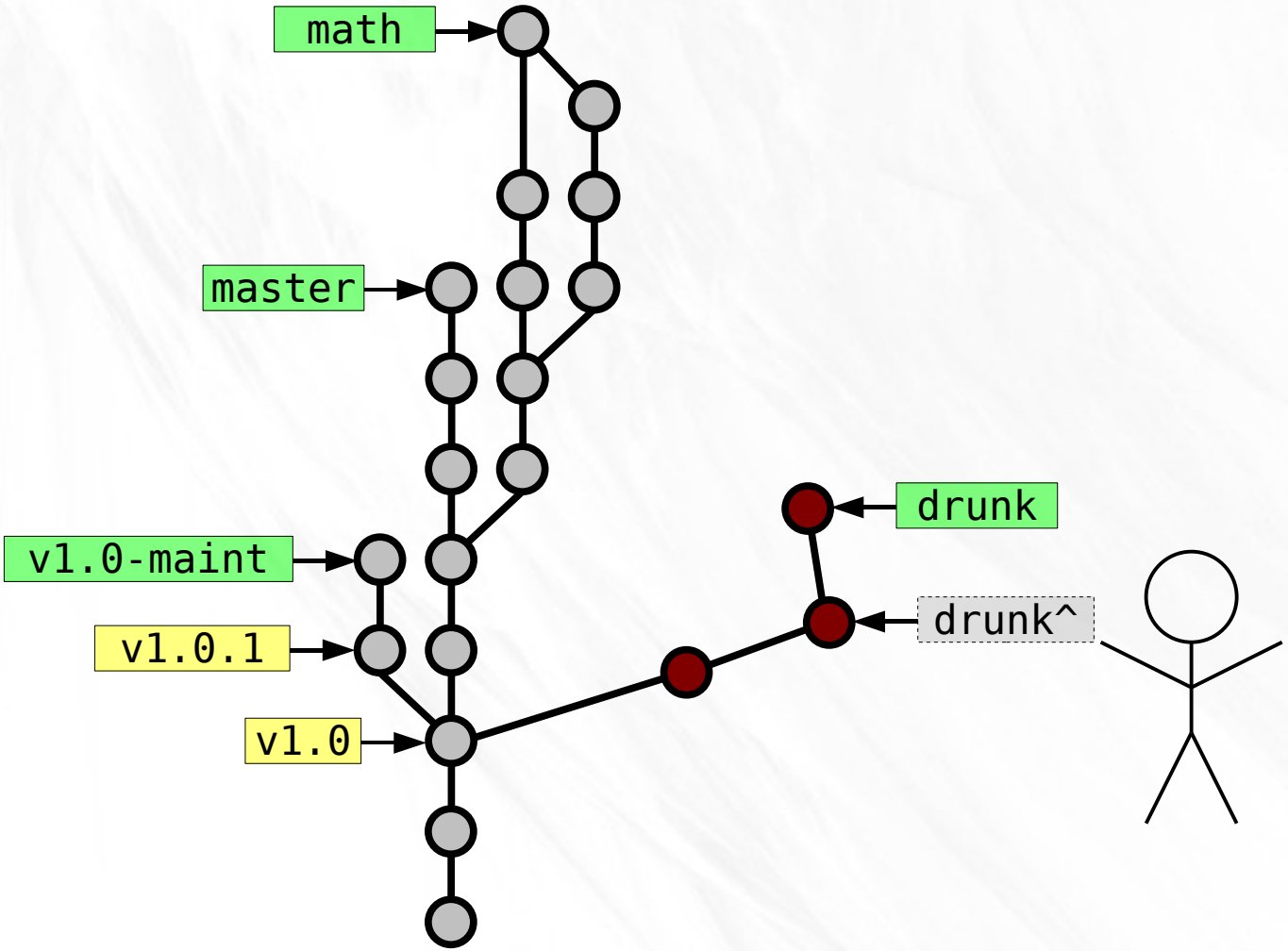


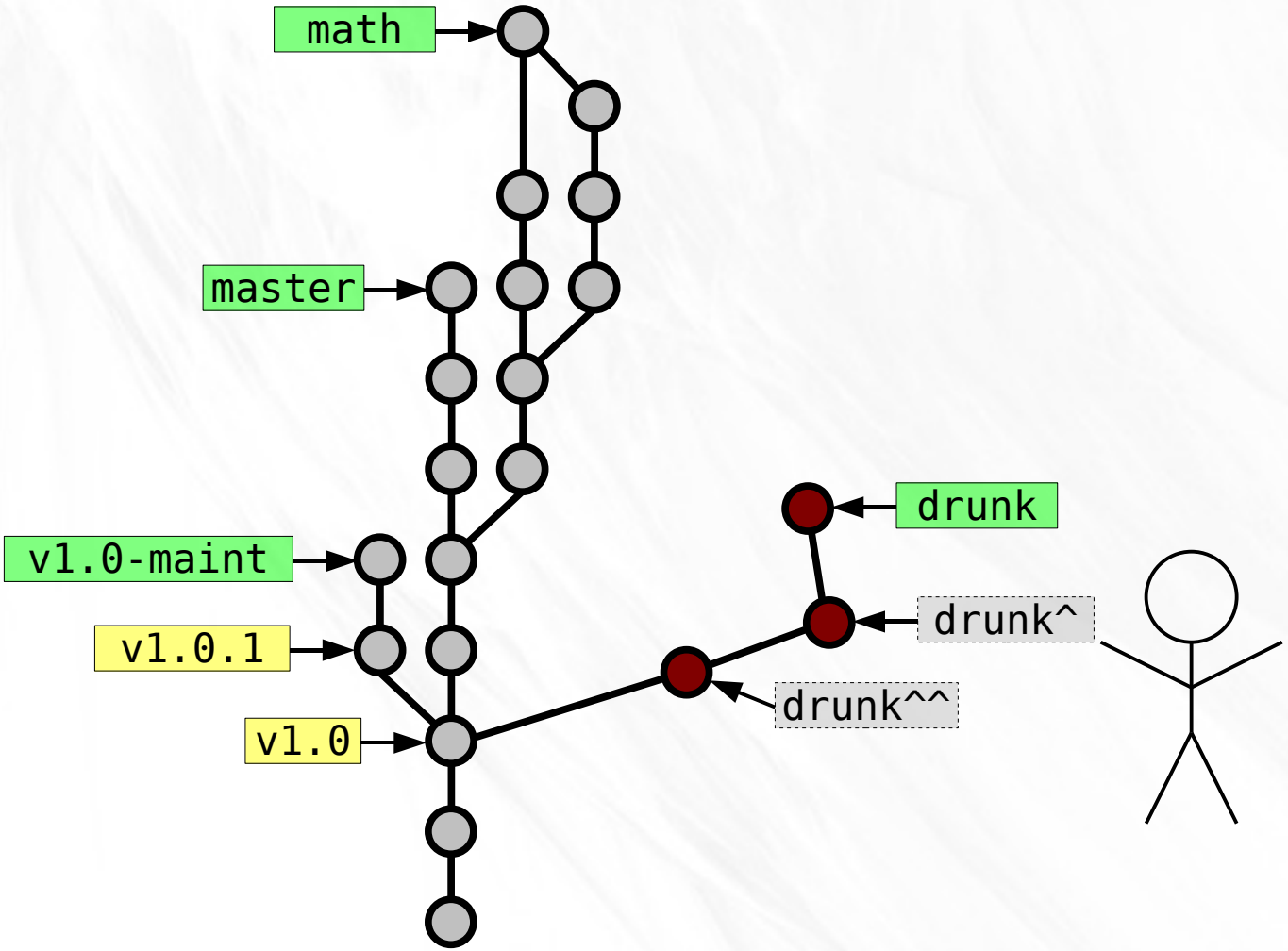
# Rewriting History



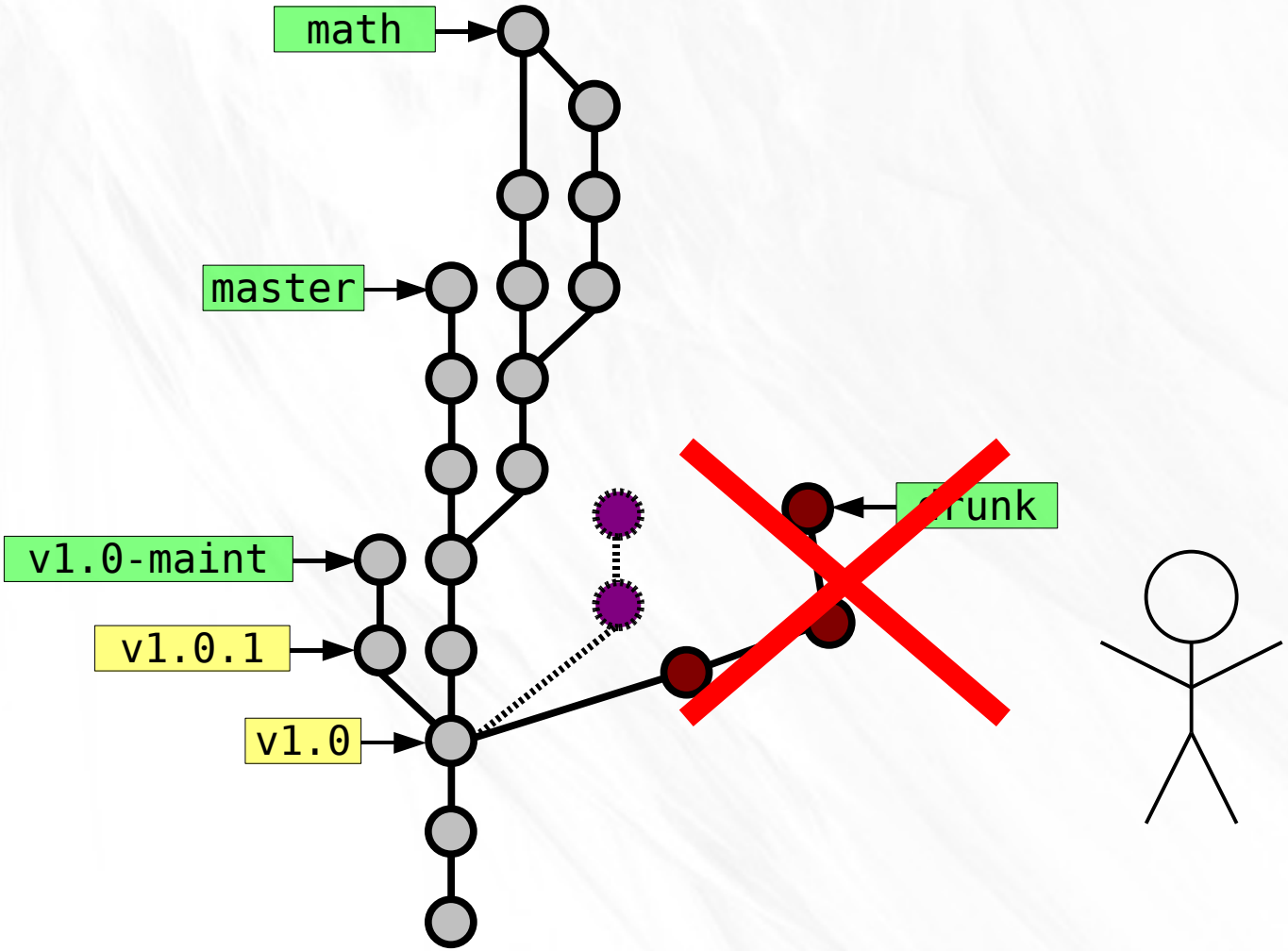
# Rewriting History



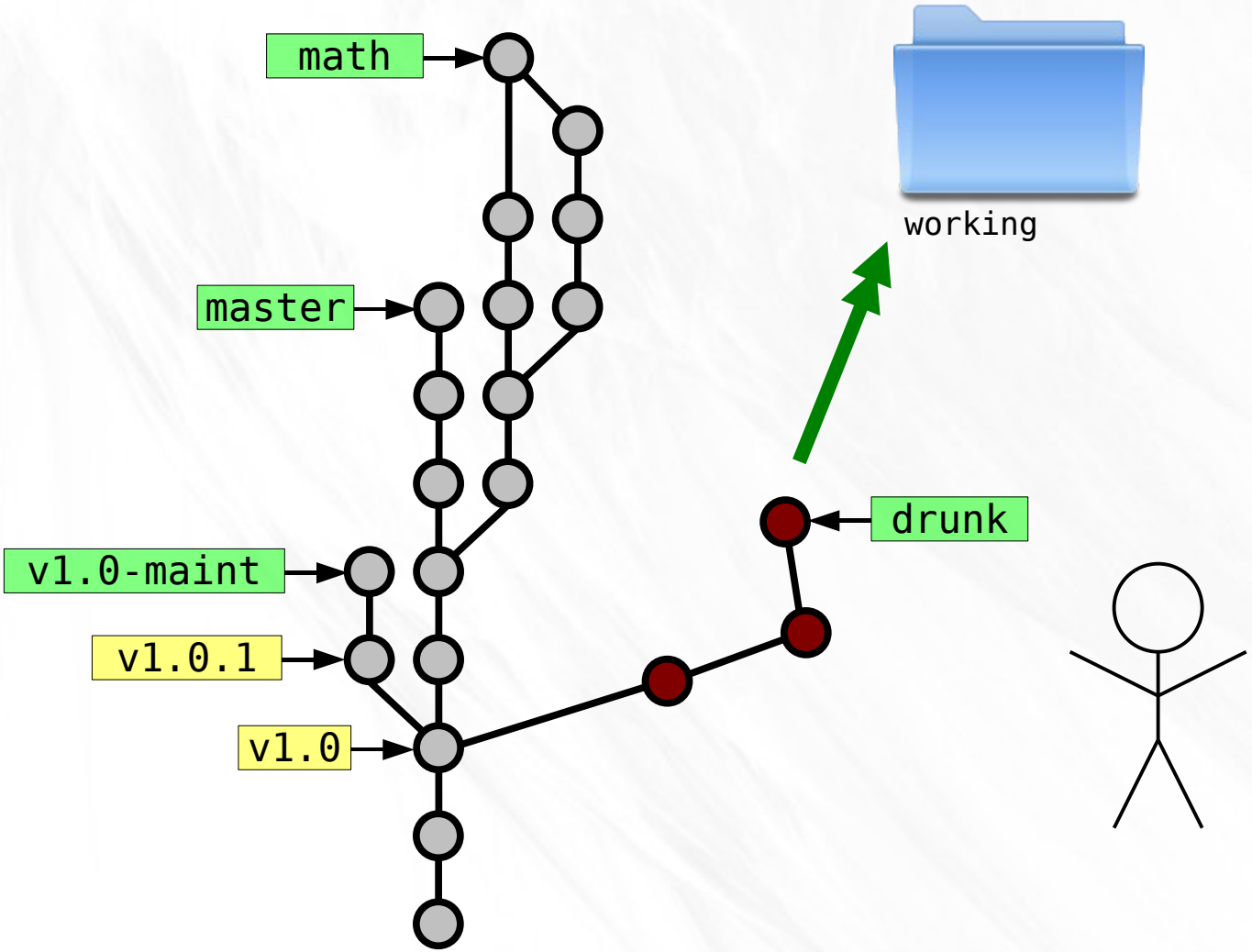
# Rewriting History



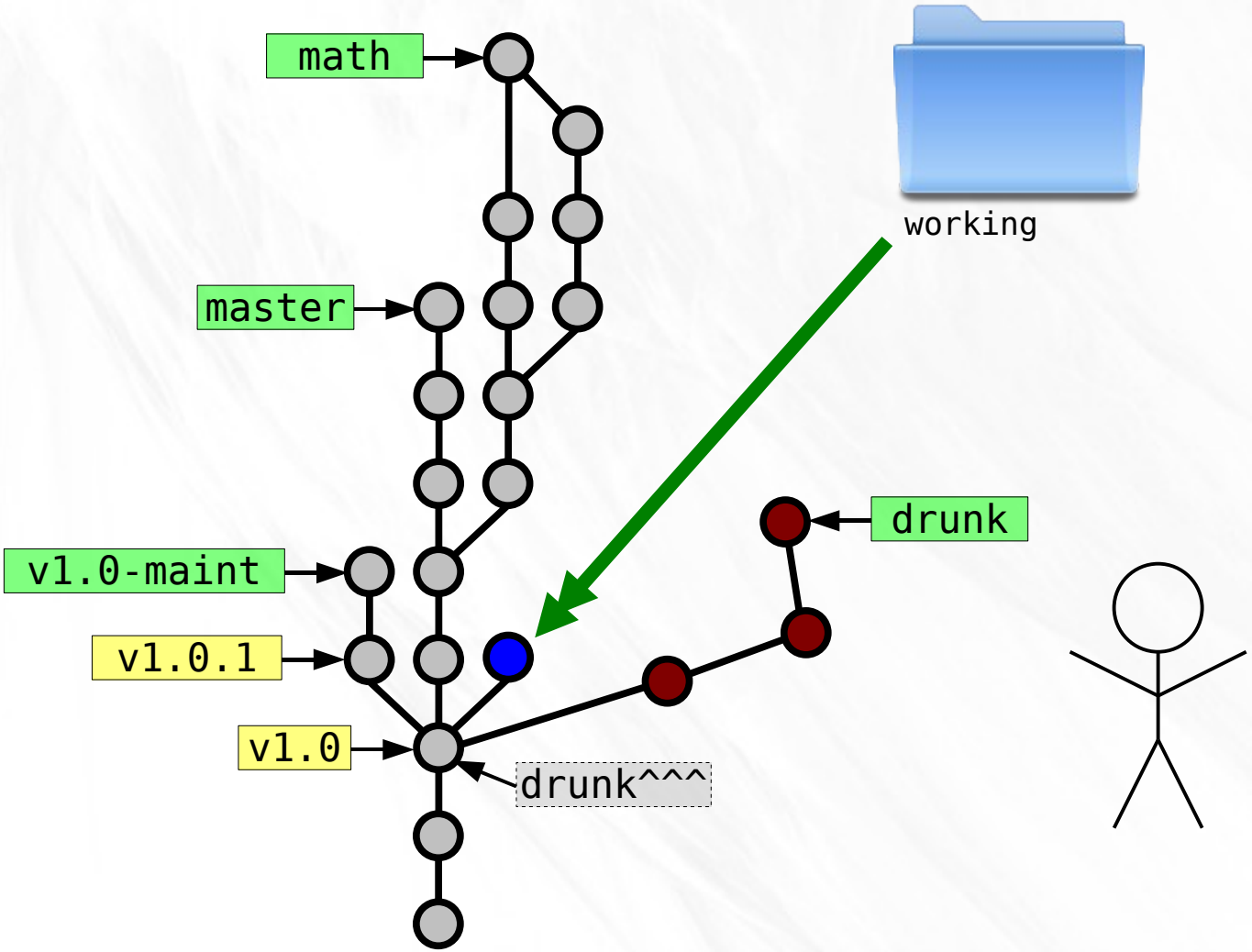
# Rewriting History



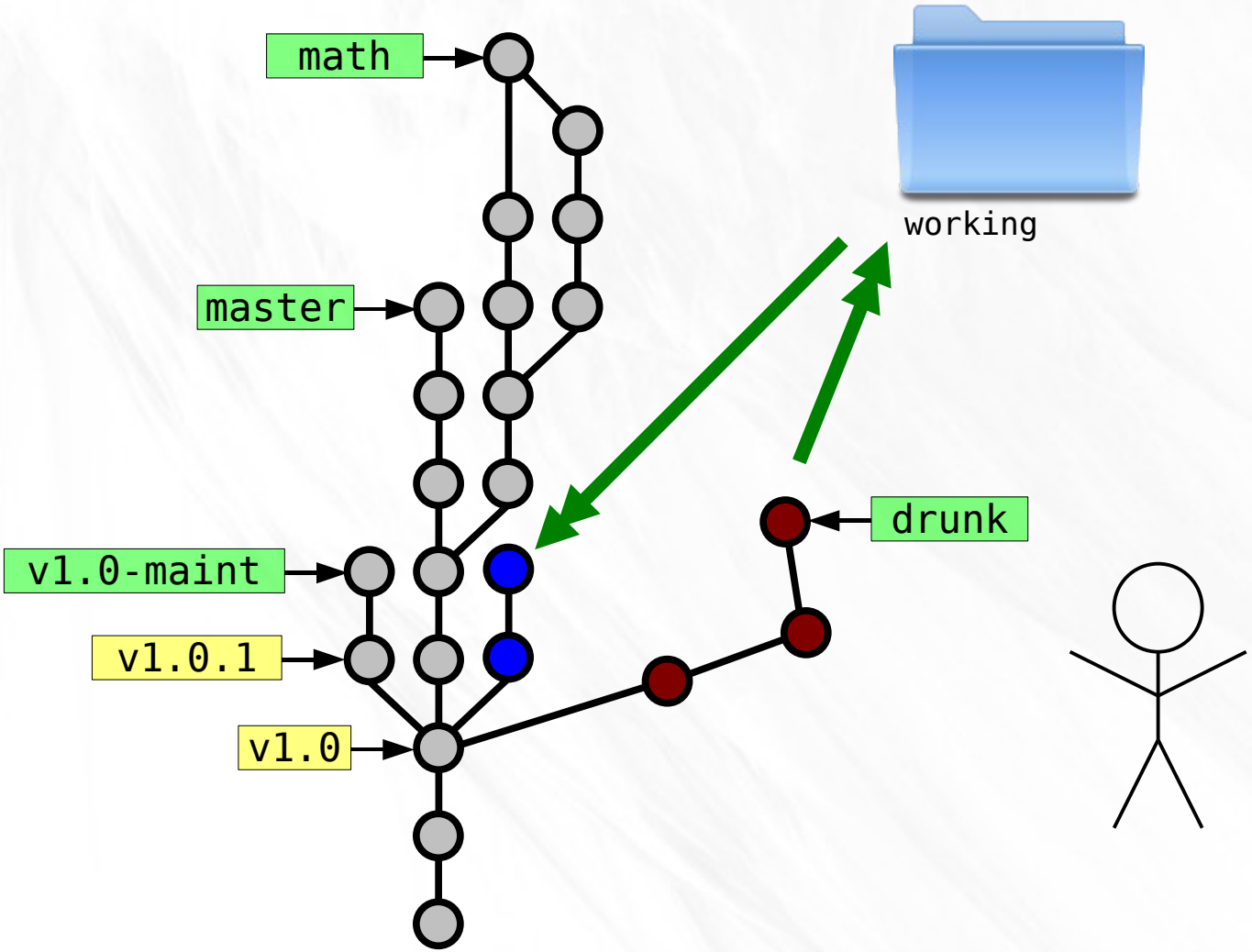
# Rewriting History



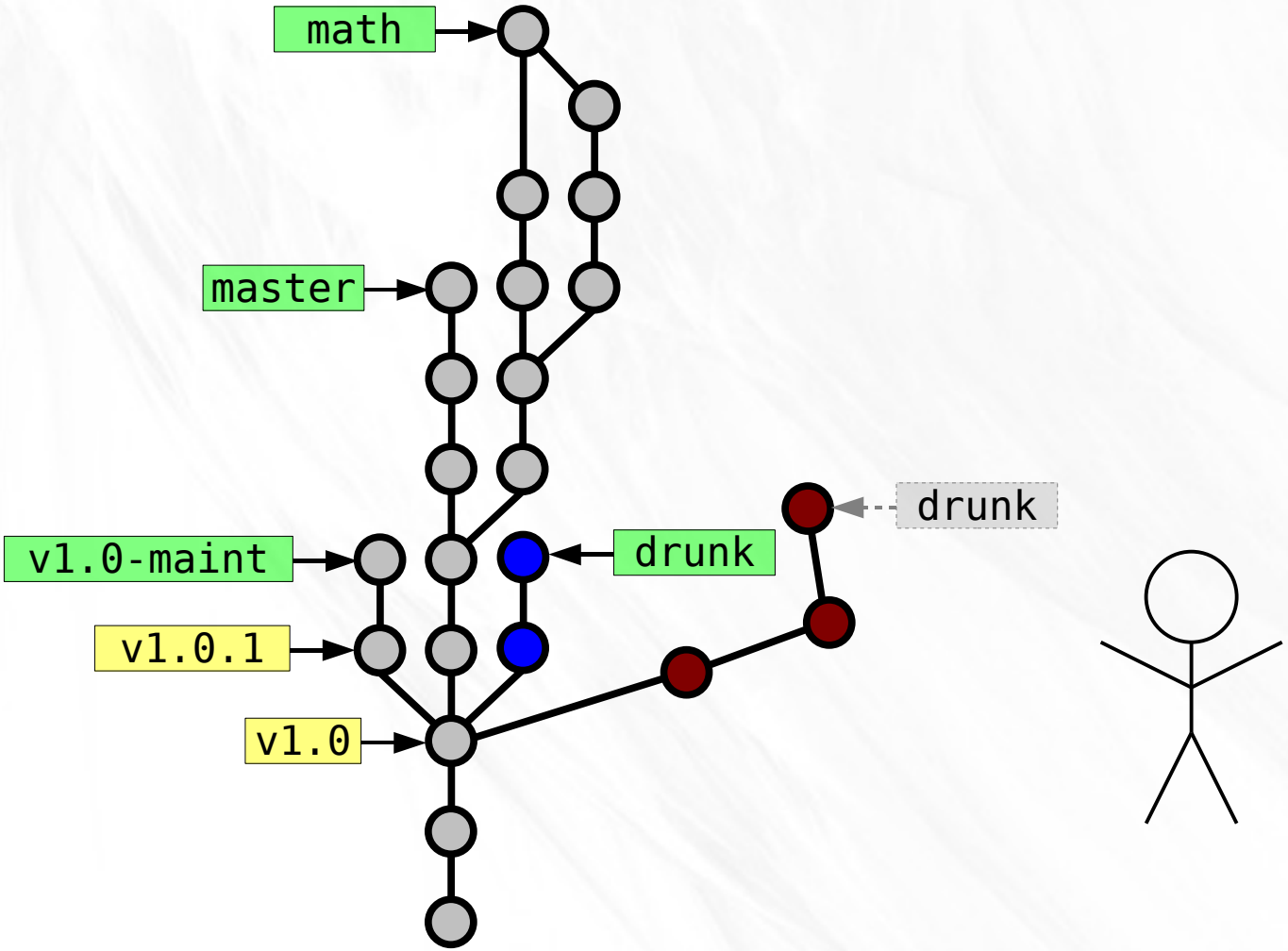
# Rewriting History



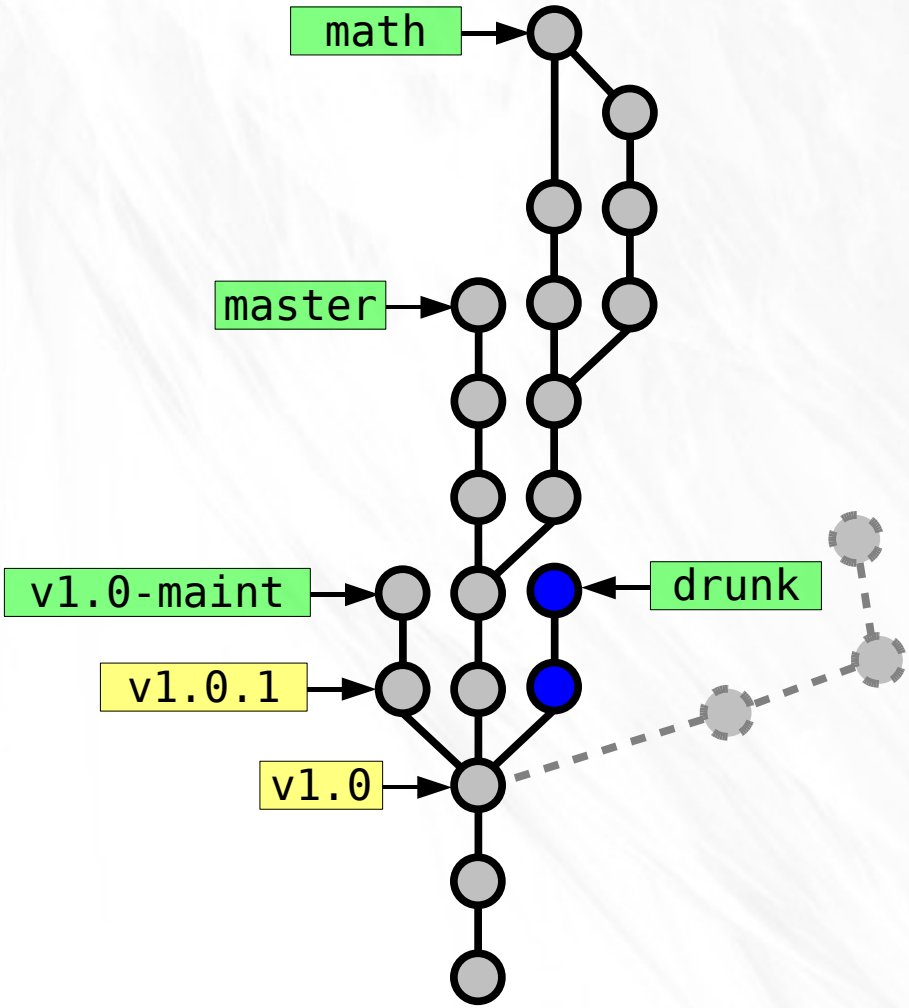
# Rewriting History

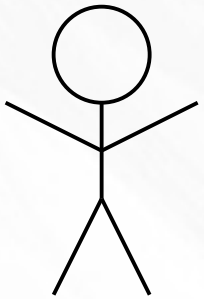


# Rewriting History

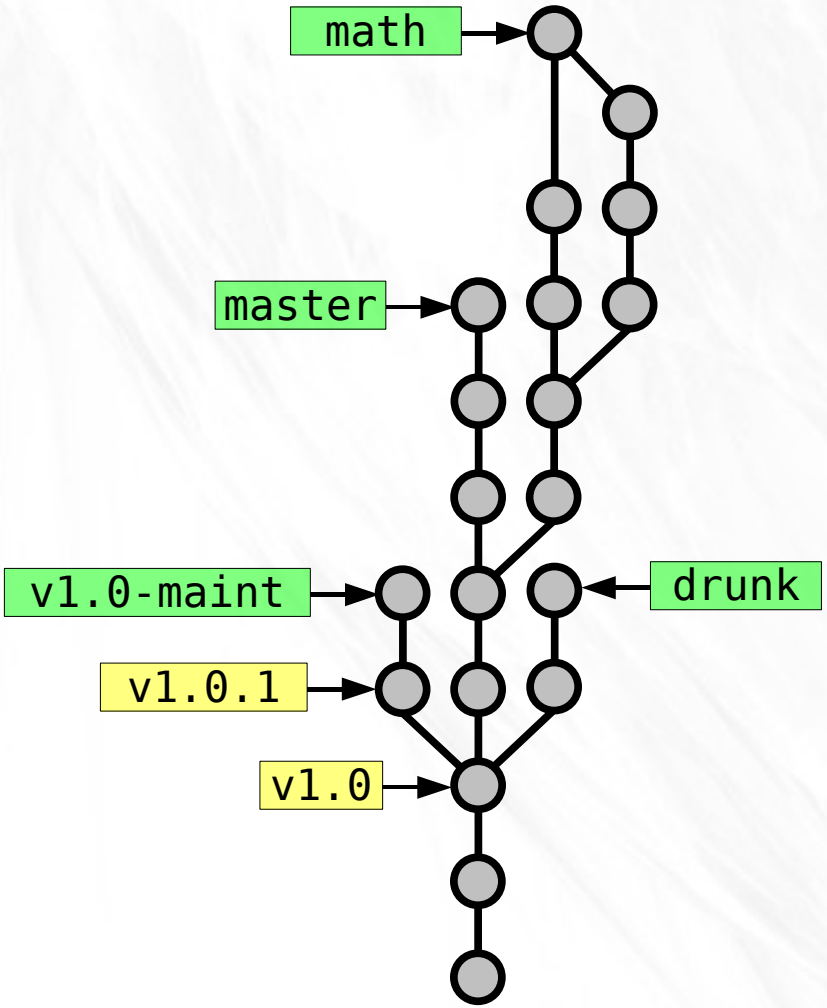


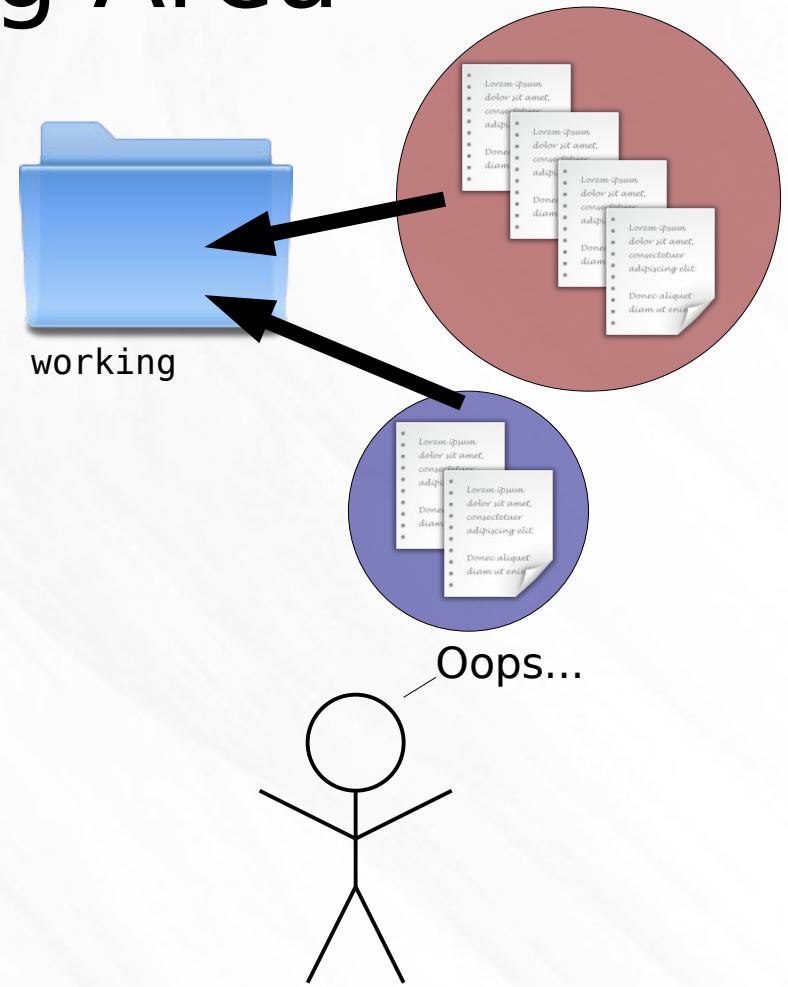
# Rewriting History



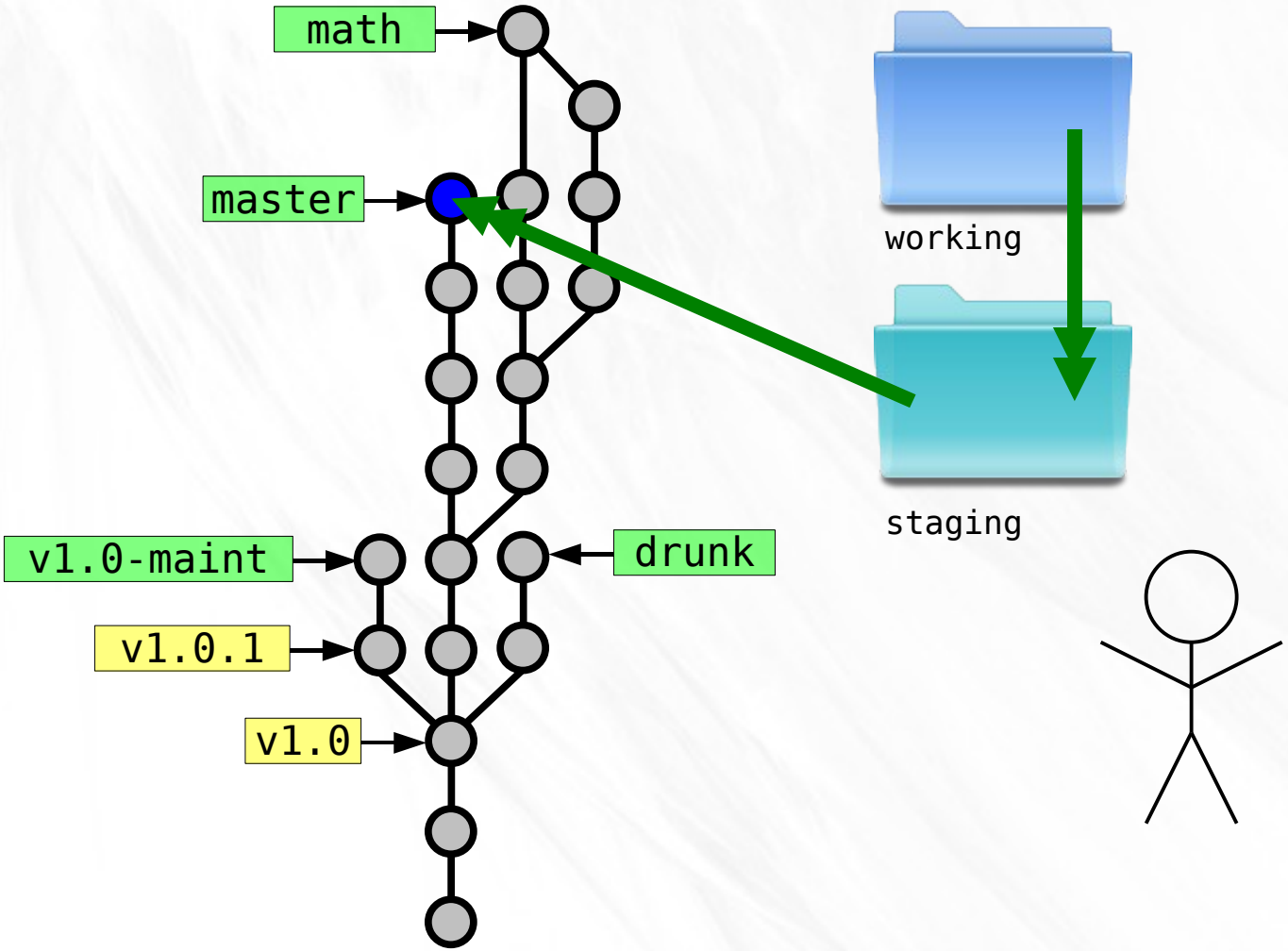



# Staging Area

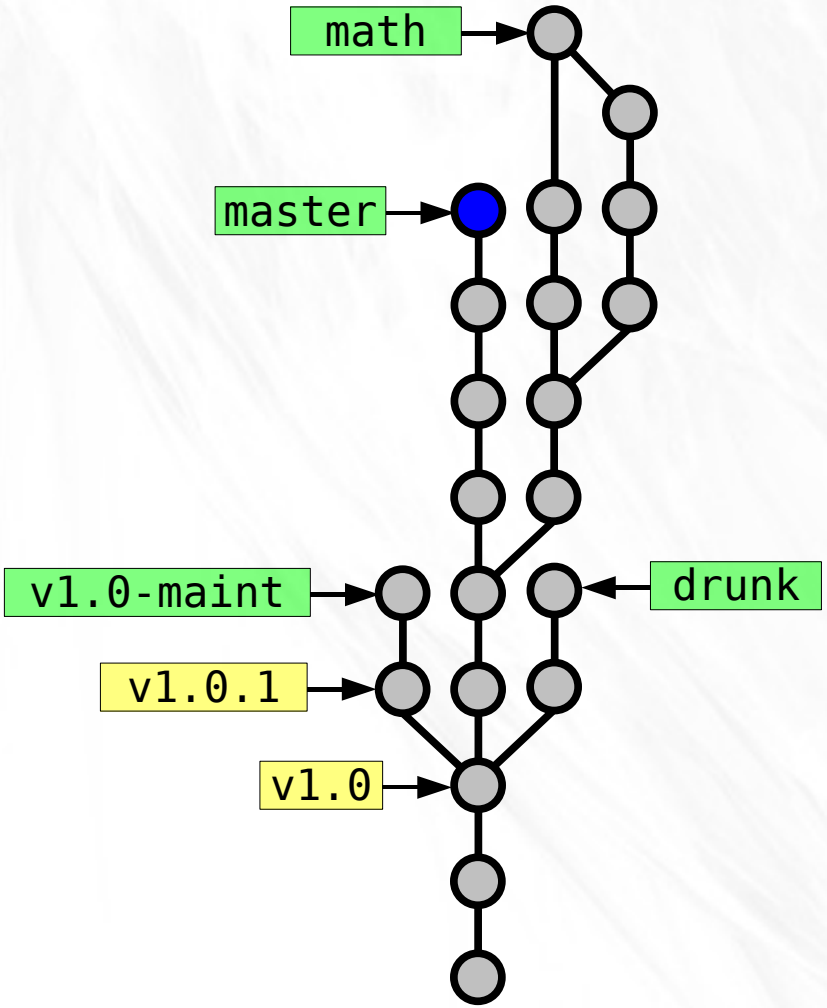


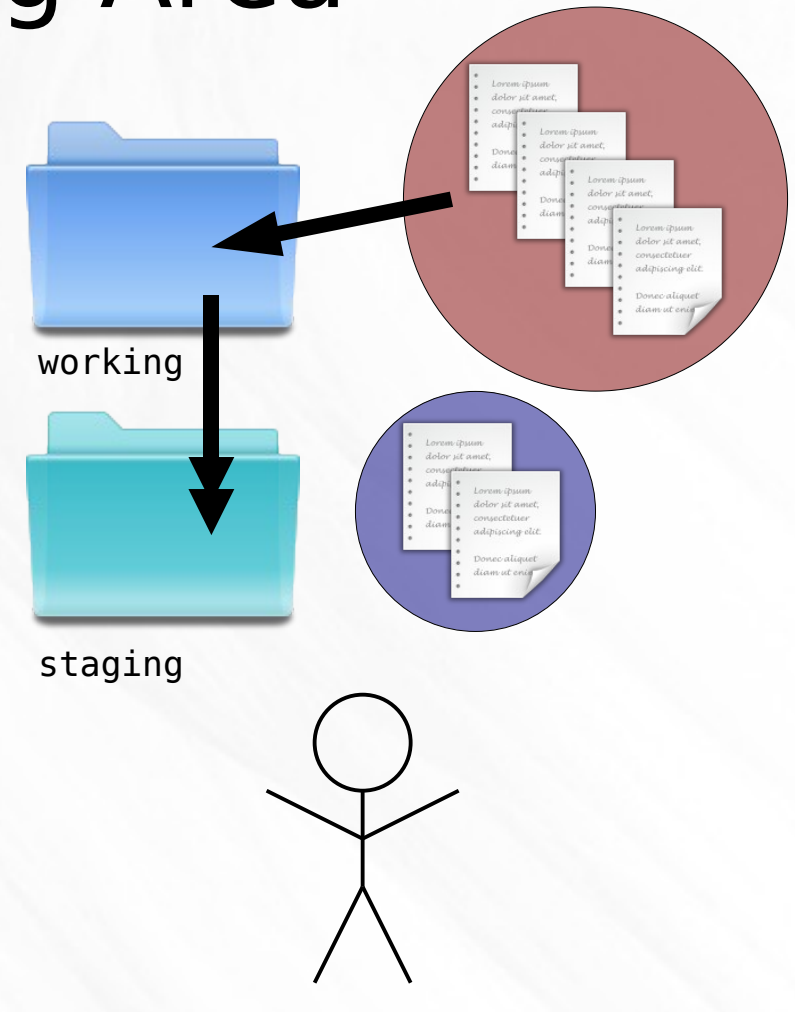



# Staging Area

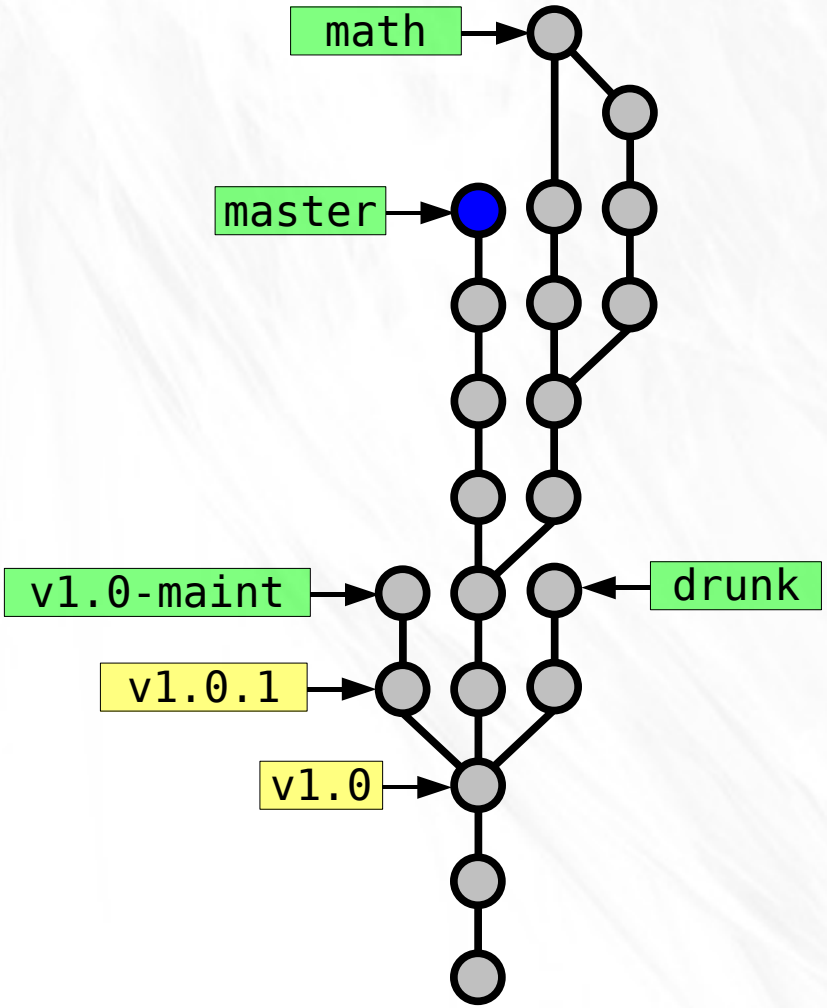


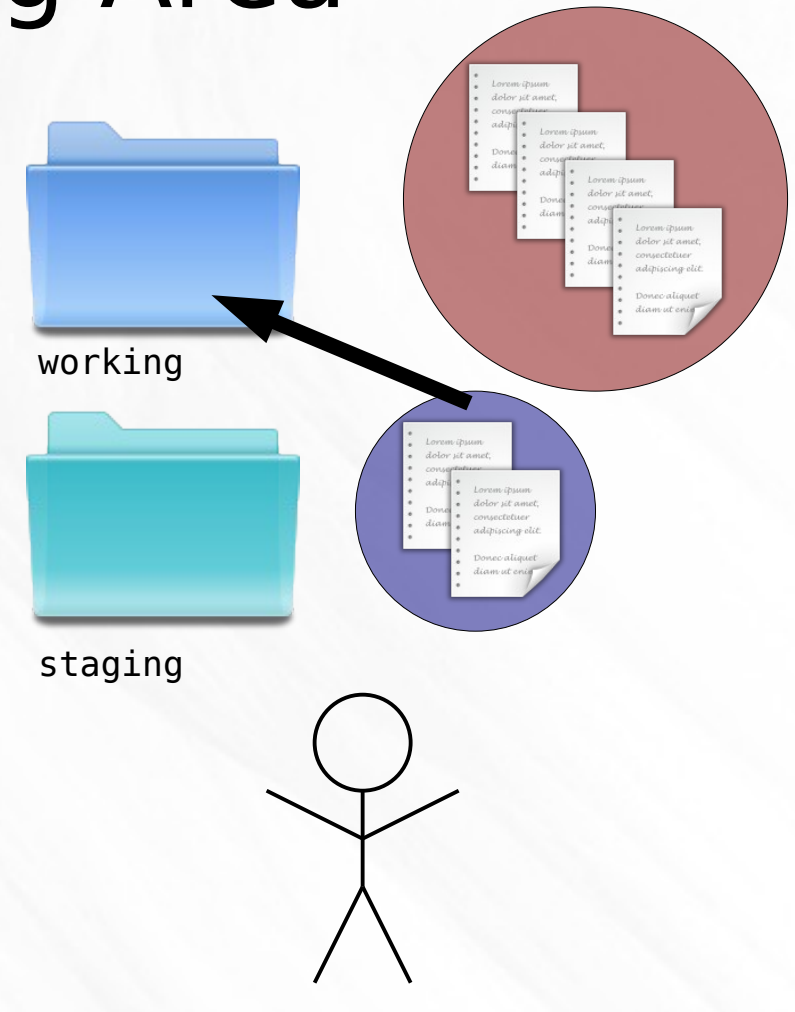
# Staging Area



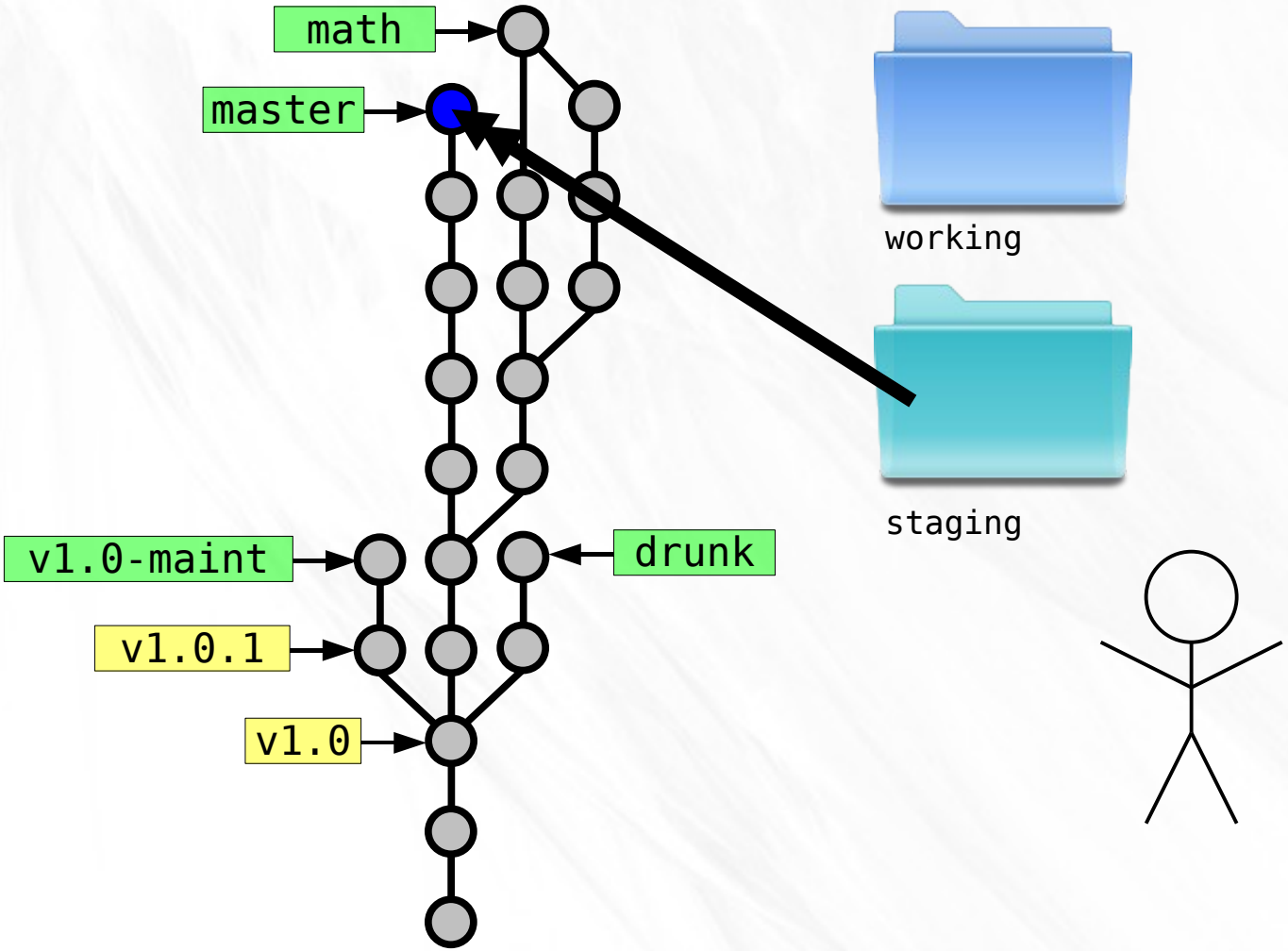



# Staging Area

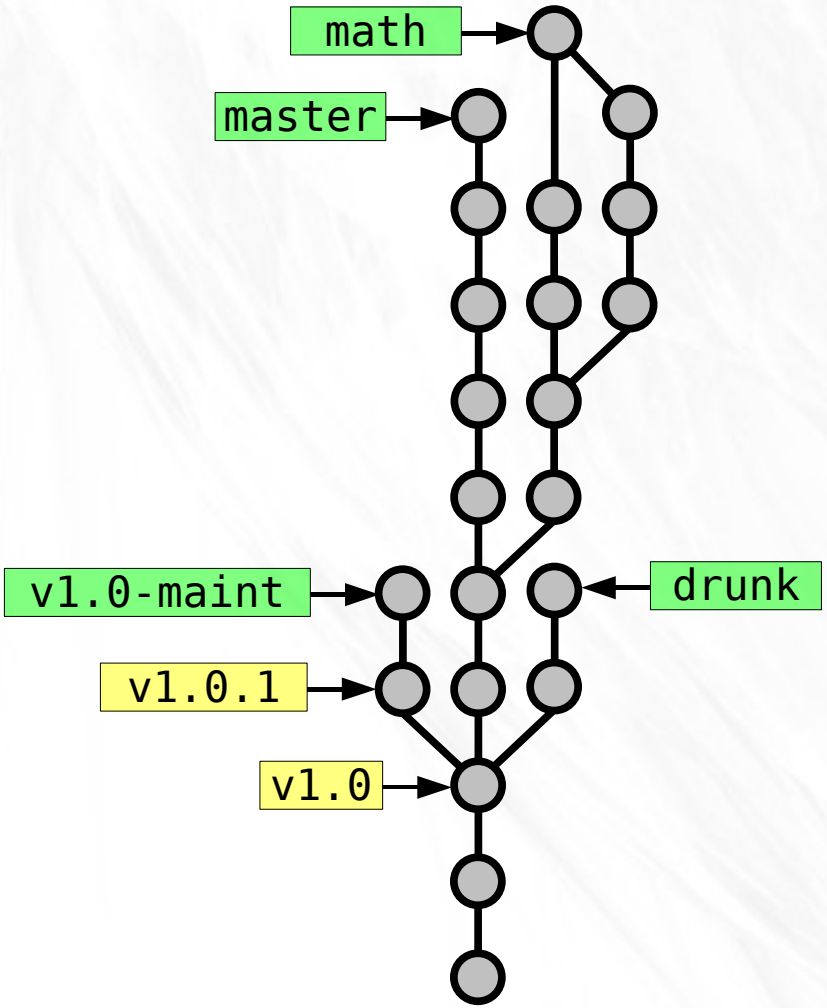


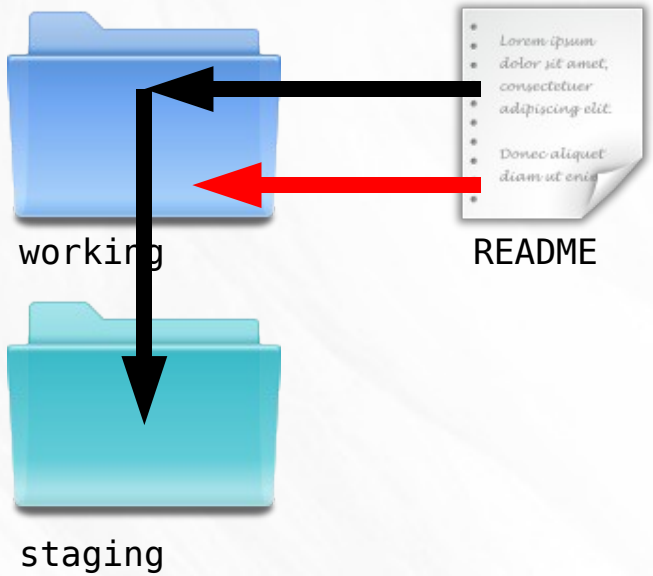



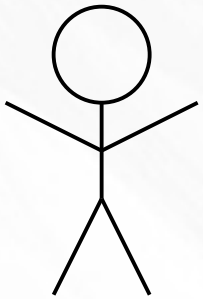
# Staging Area



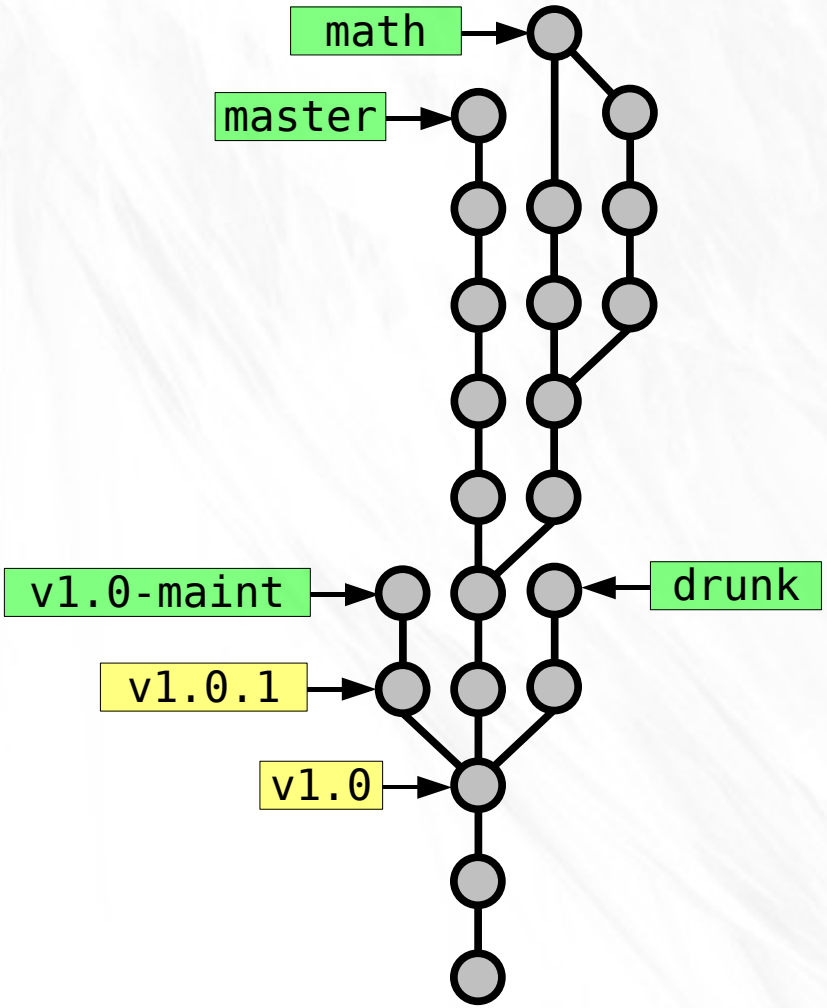
# Staging Area



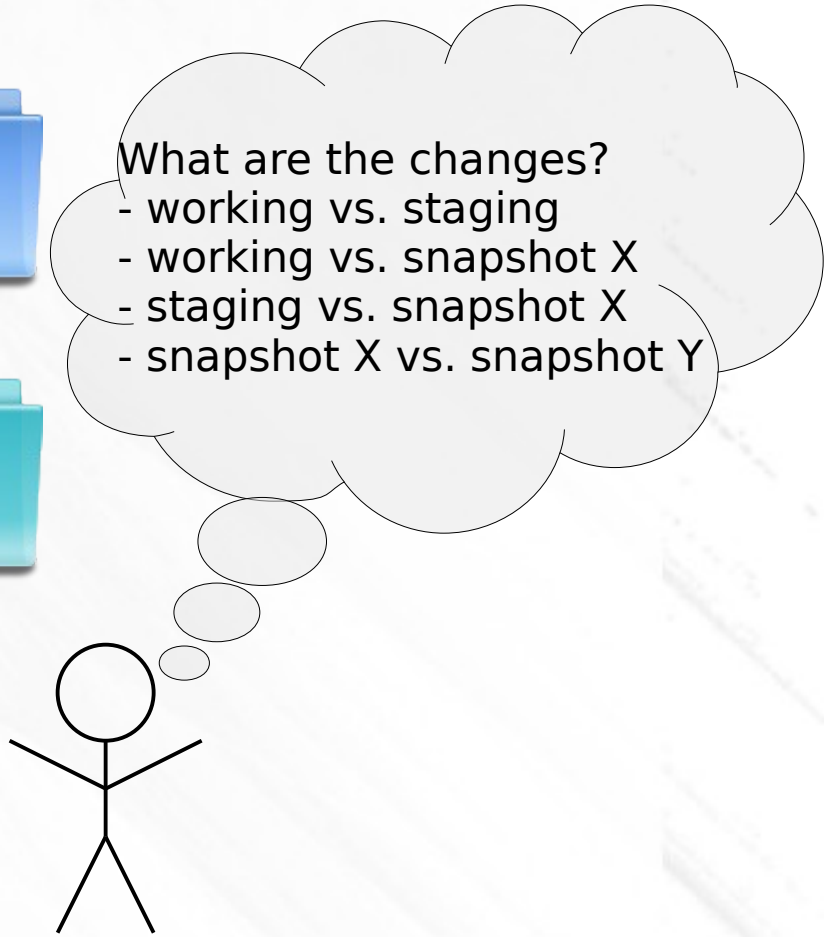



# Diffs

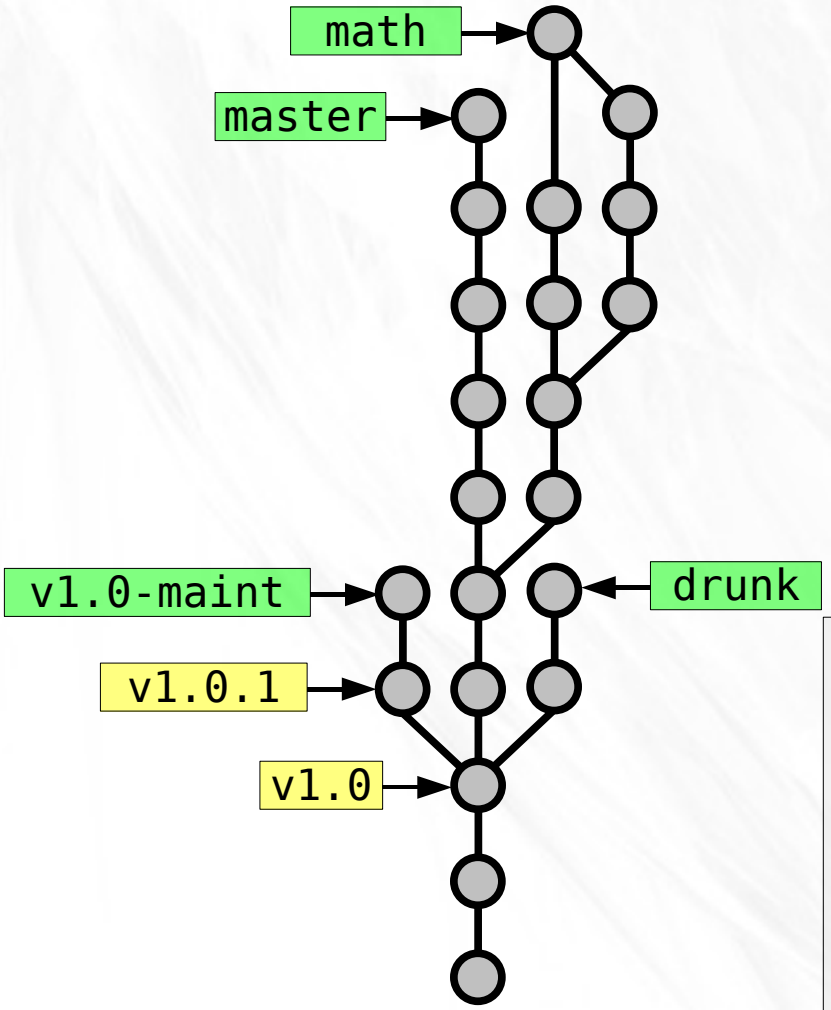


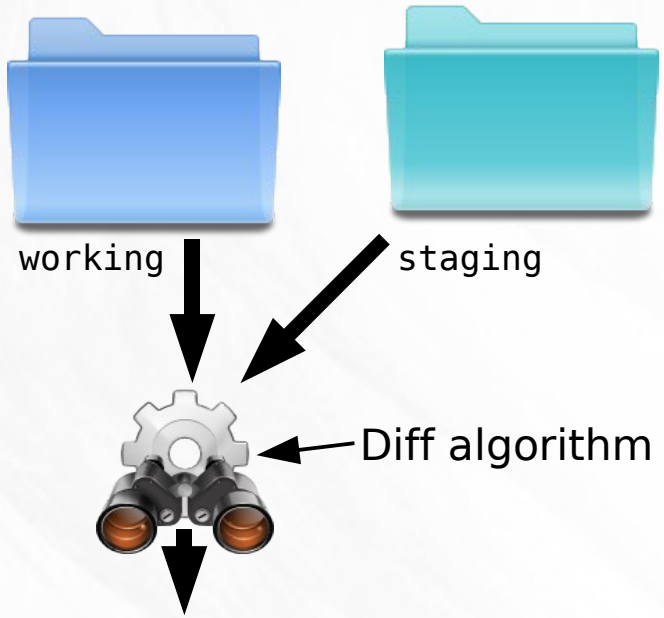

working

staging

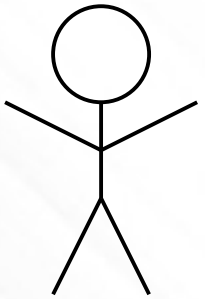


# Diffs

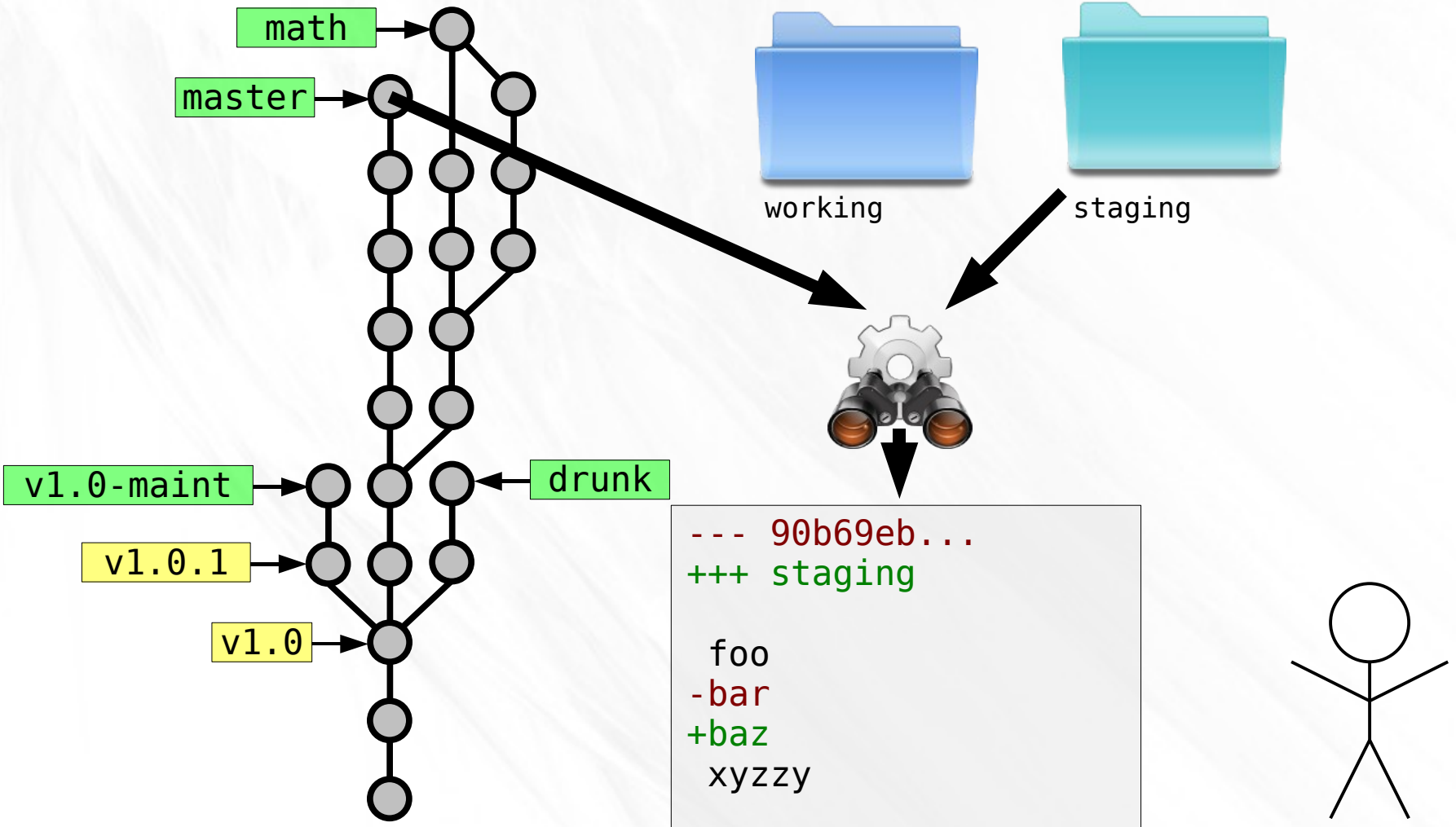


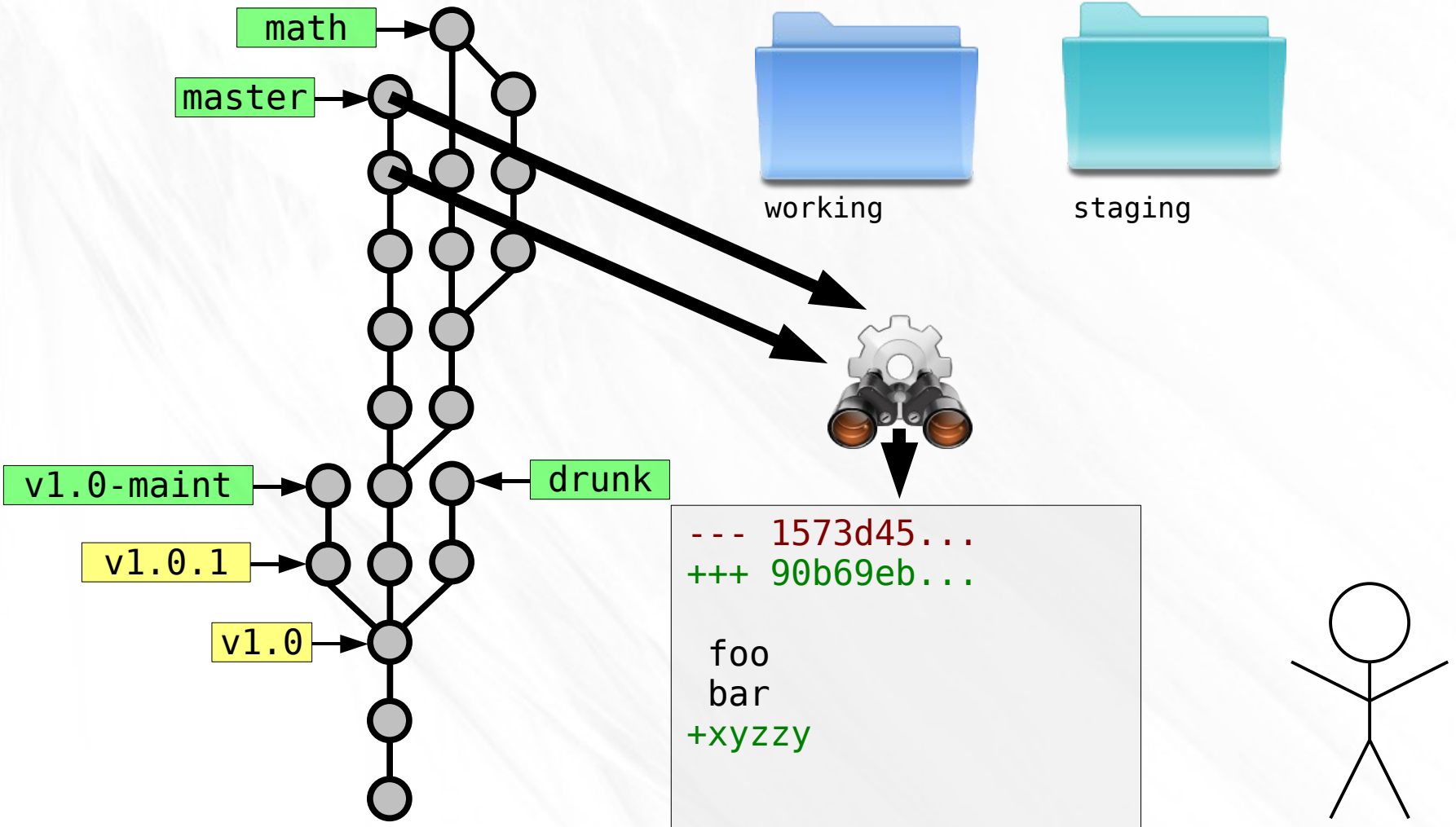

```
--- staging
+++ working

 foo
-baz
+bazzle
 xyzzy
```



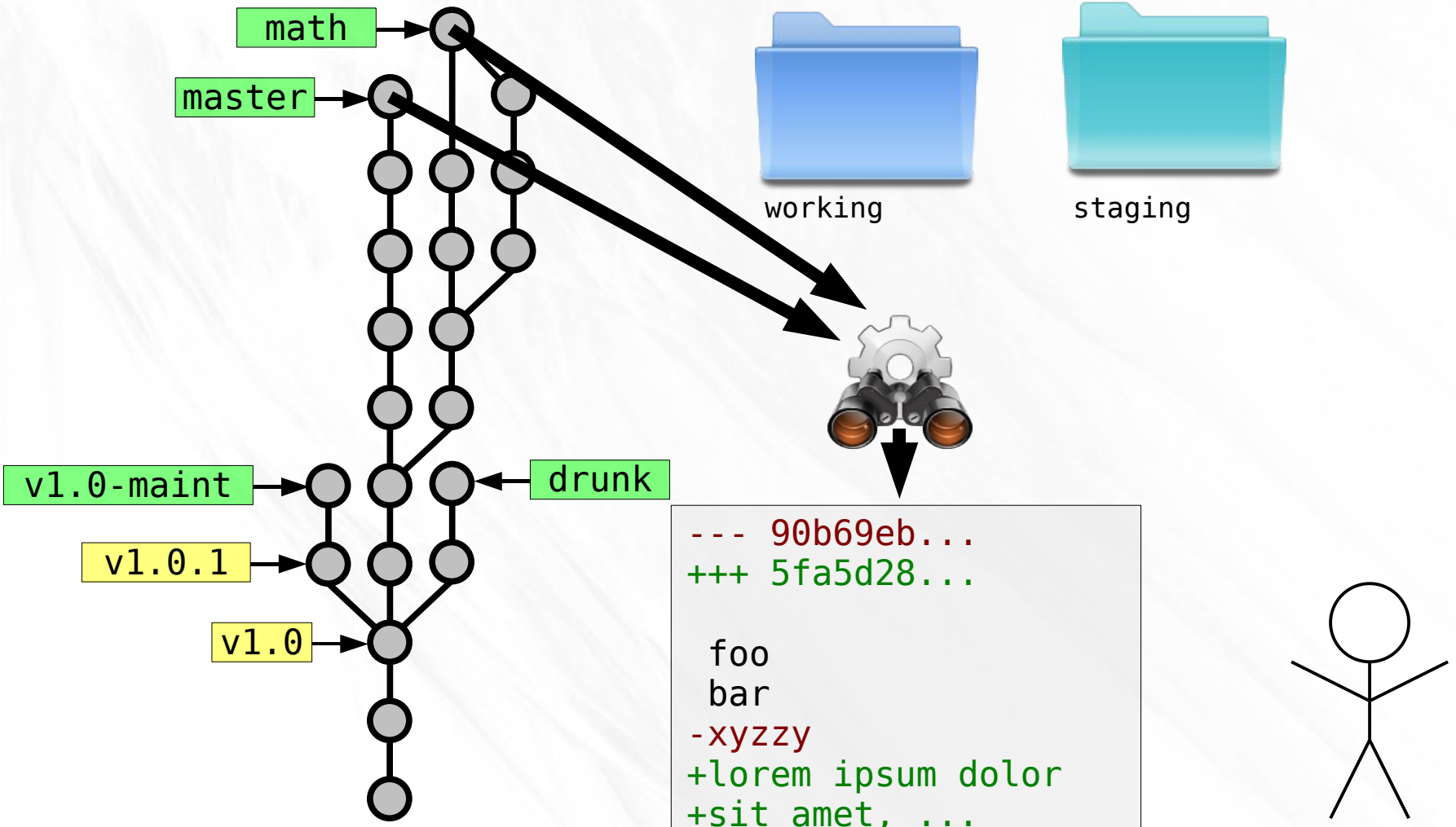
# Diffs


math
master
v1.0-maint
drunk
v1.0.1
v1.0
working
staging
--- 90b69eb...
+++ staging
foo
-bar
+baz
xyzzy

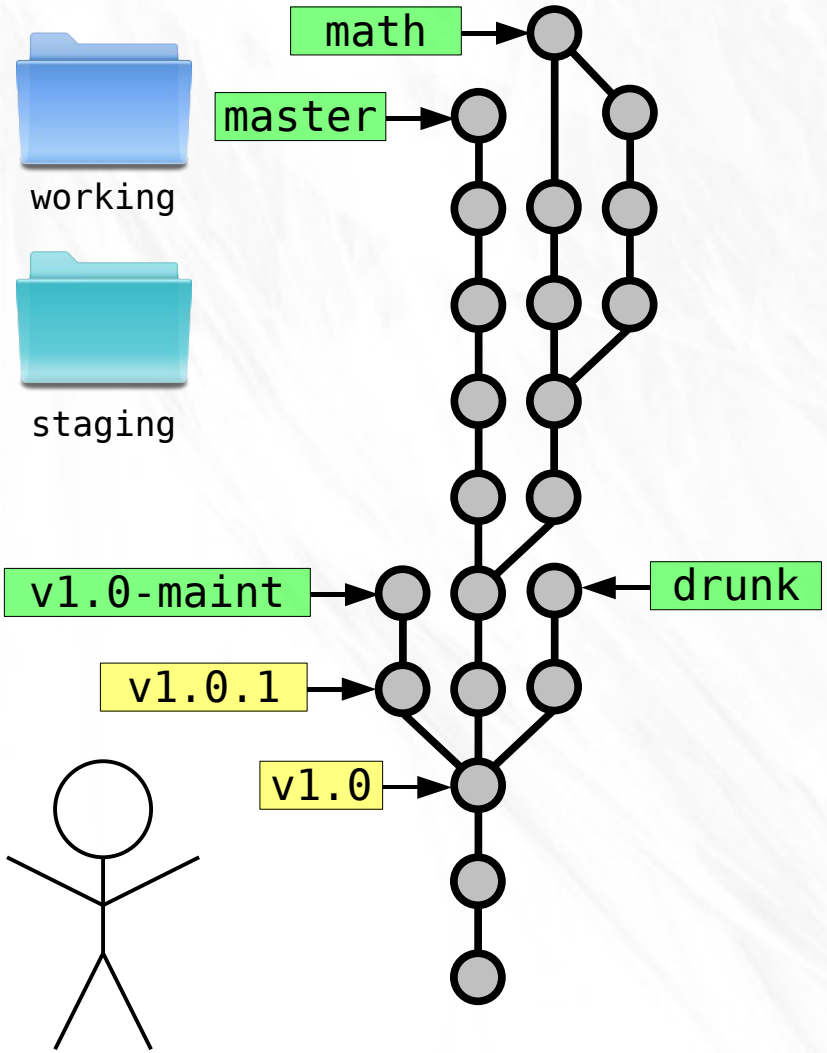
# Diffs

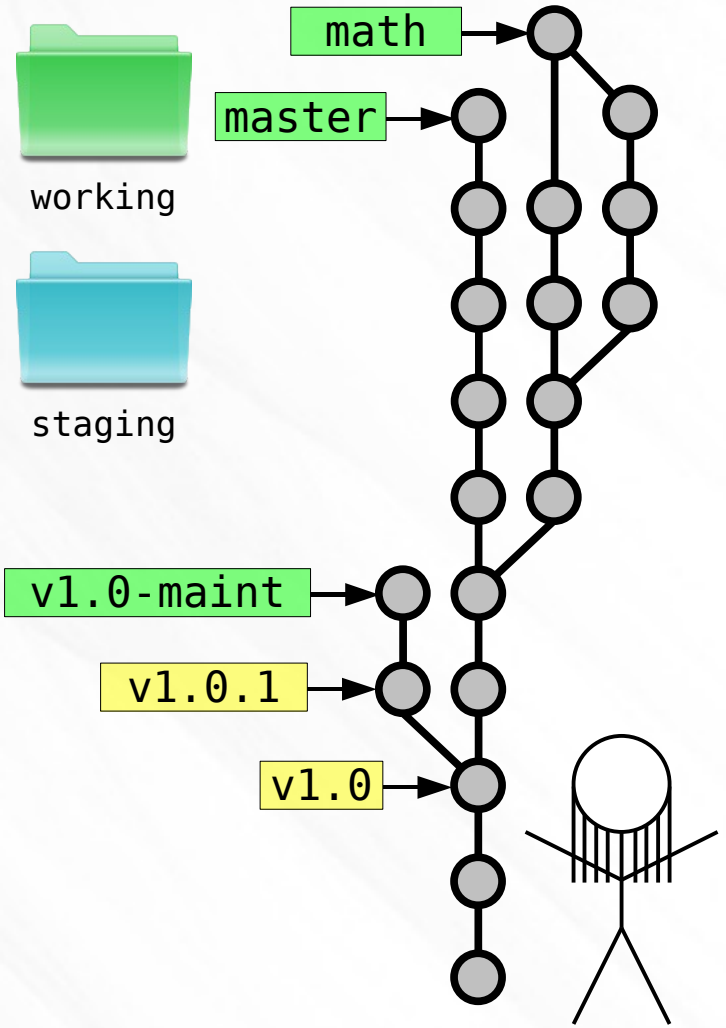

math
master
v1.0-maint
drunk
v1.0.1
v1.0
working
staging
--- 1573d45...
+++ 90b69eb...
foo
bar
+xyzzy

# Diffs

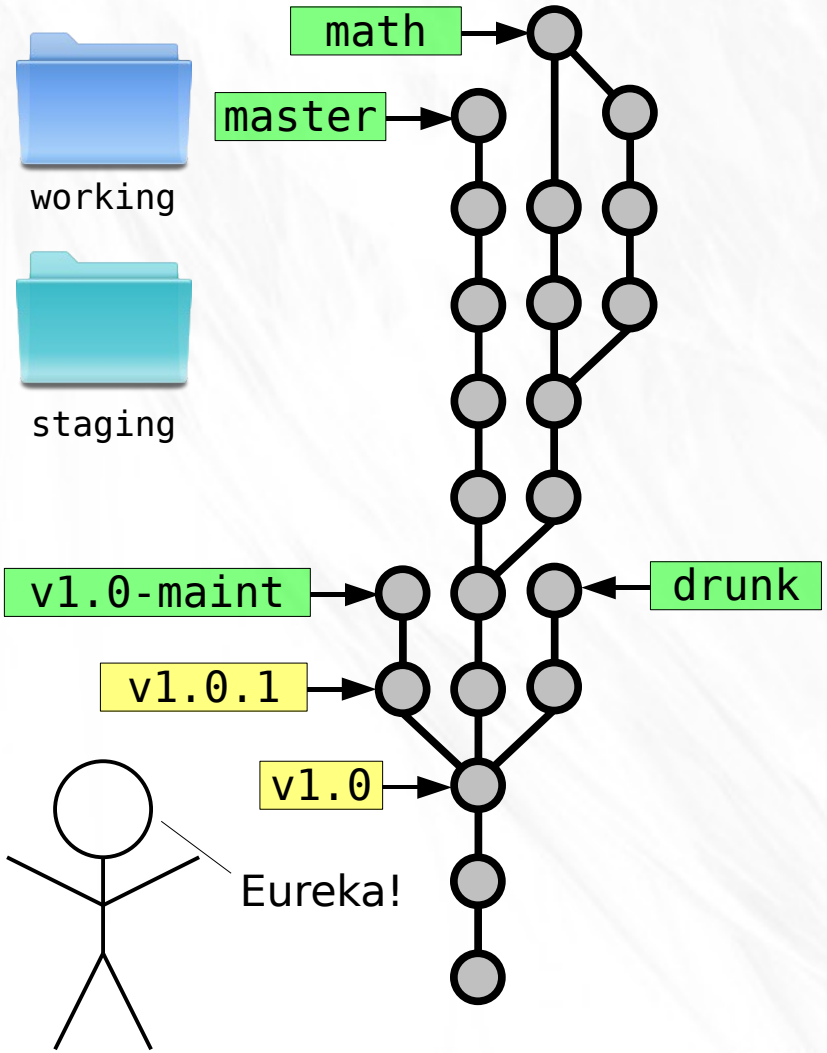

math
master
v1.0-maint
drunk
v1.0.1
v1.0
working
staging
--- 90b69eb...
+++ 5fa5d28...
foo
bar
-xyzzy
+lorem ipsum dolor
+sit amet, ...

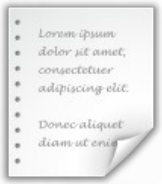
# Eliminating Duplication



# Eliminating Duplication



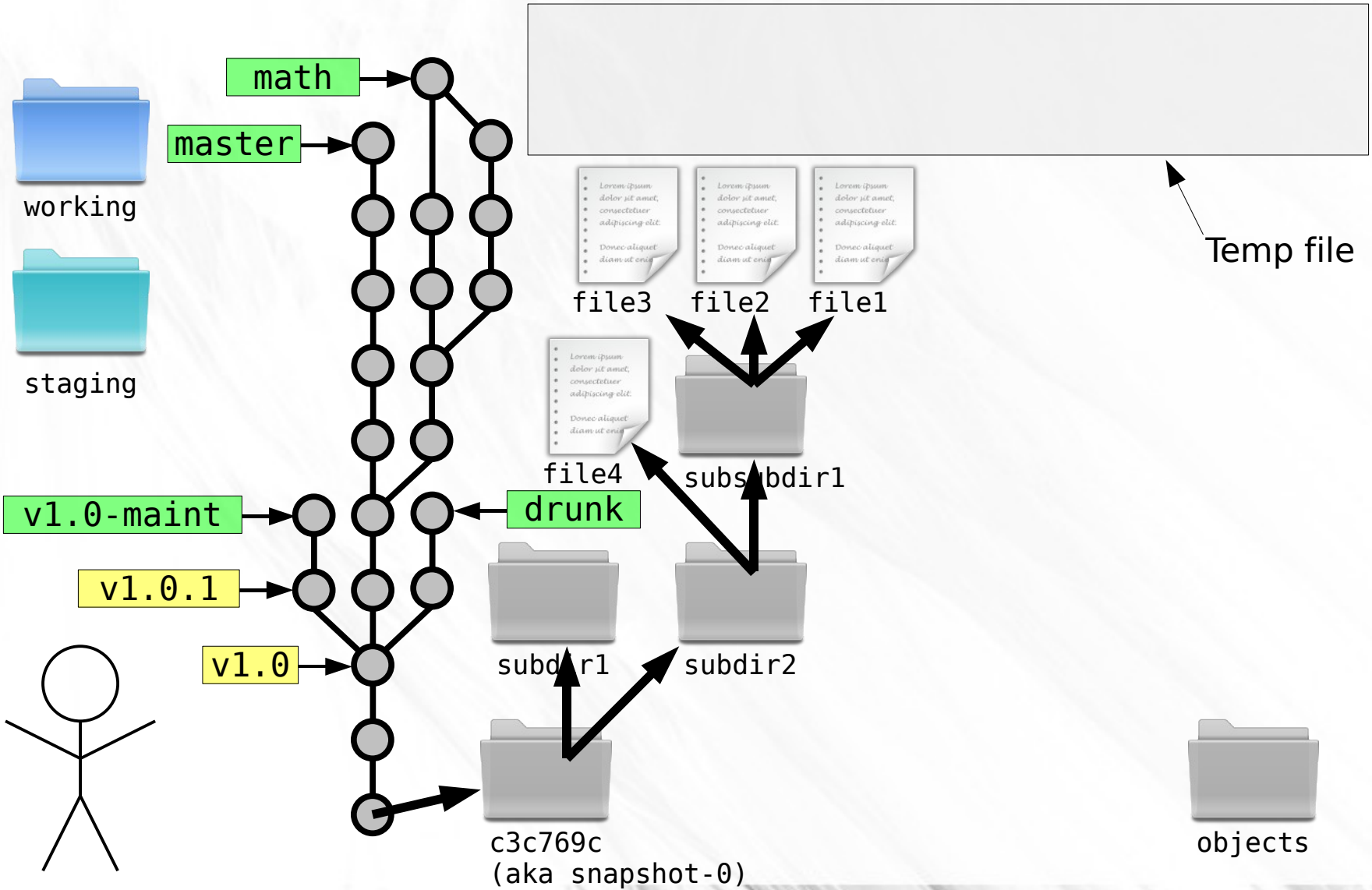



SHA1

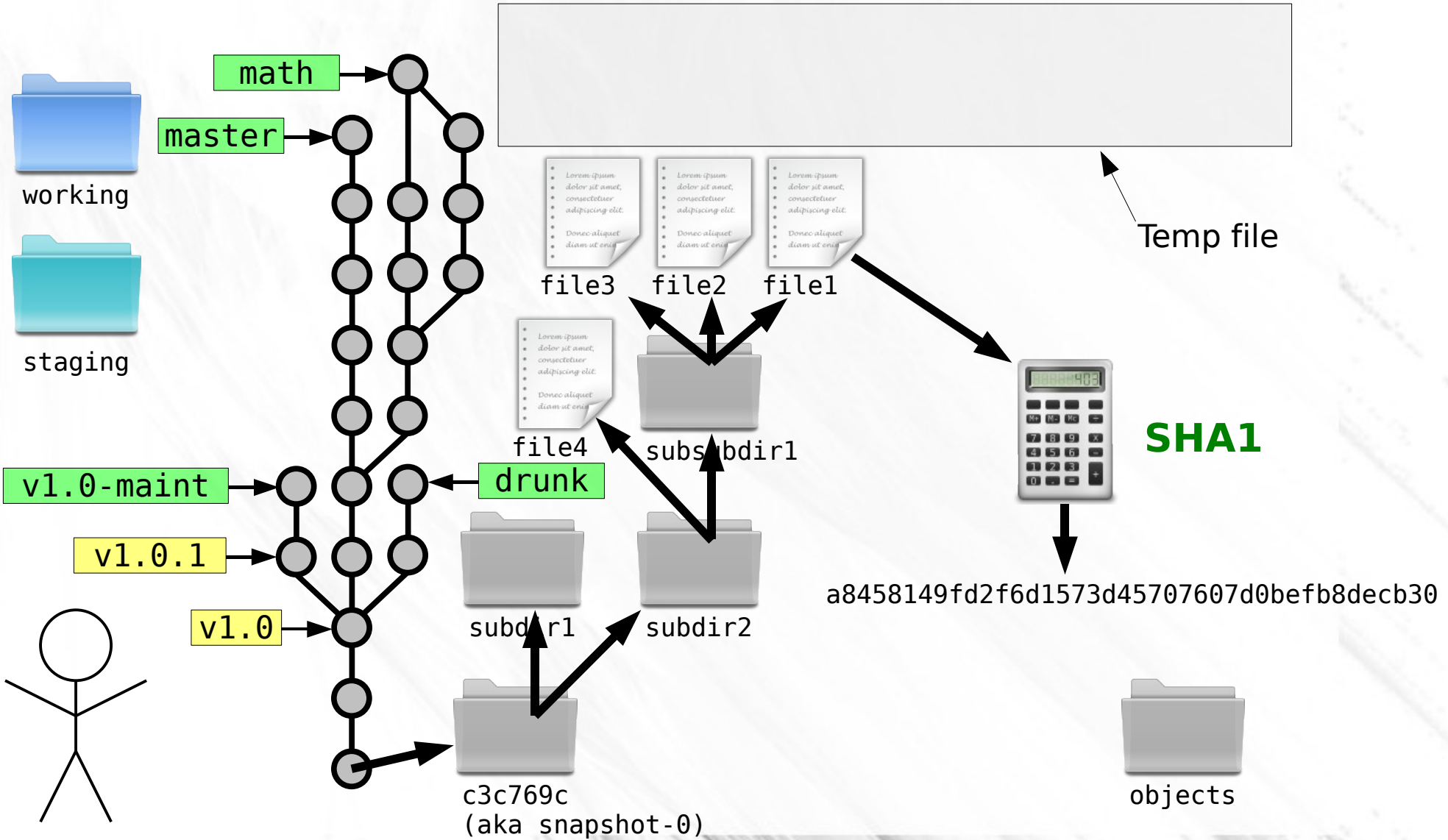
2804133755c3ed396d162028c7b30a1cbcfecded

objects

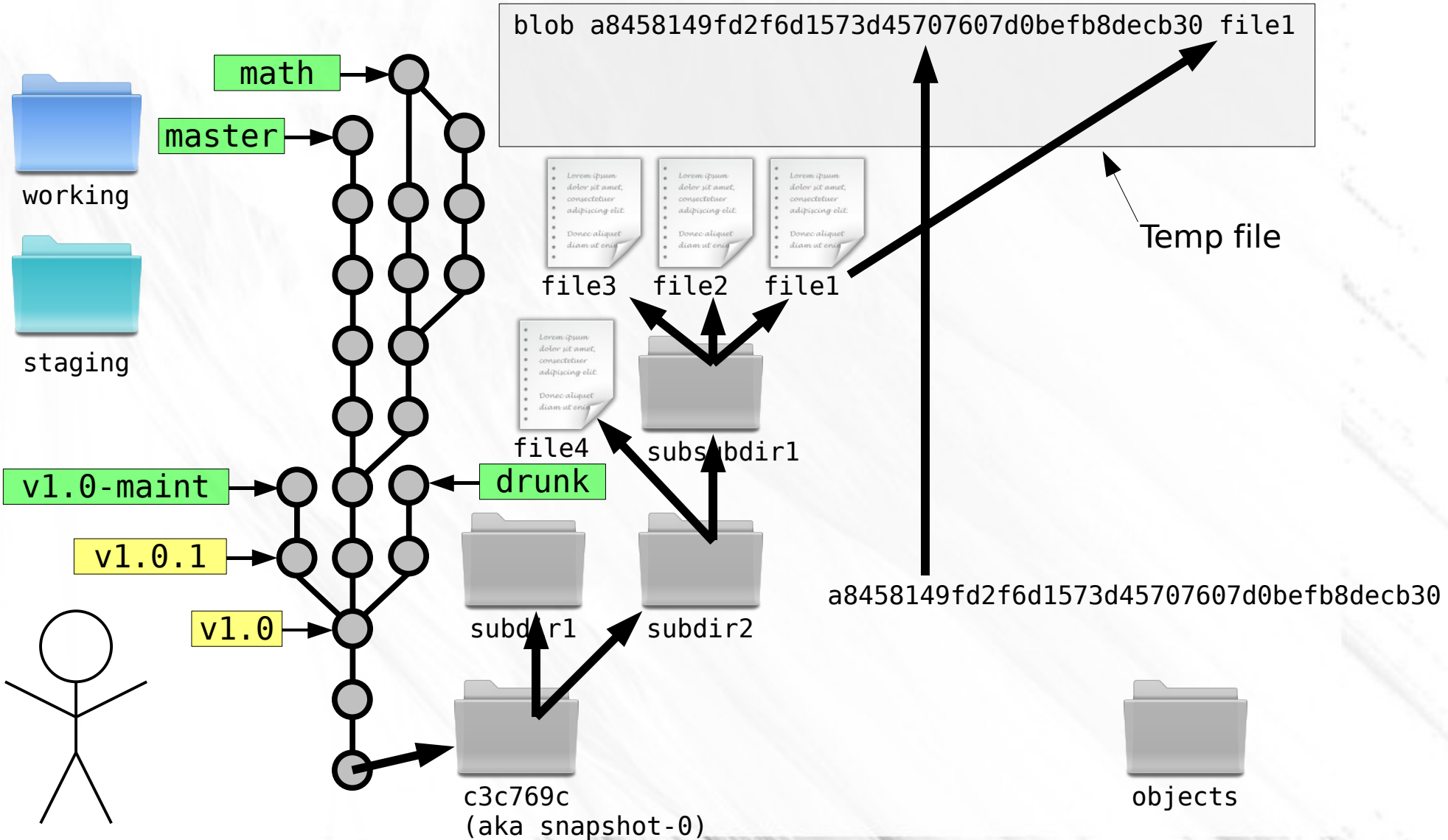
# Eliminating Duplication



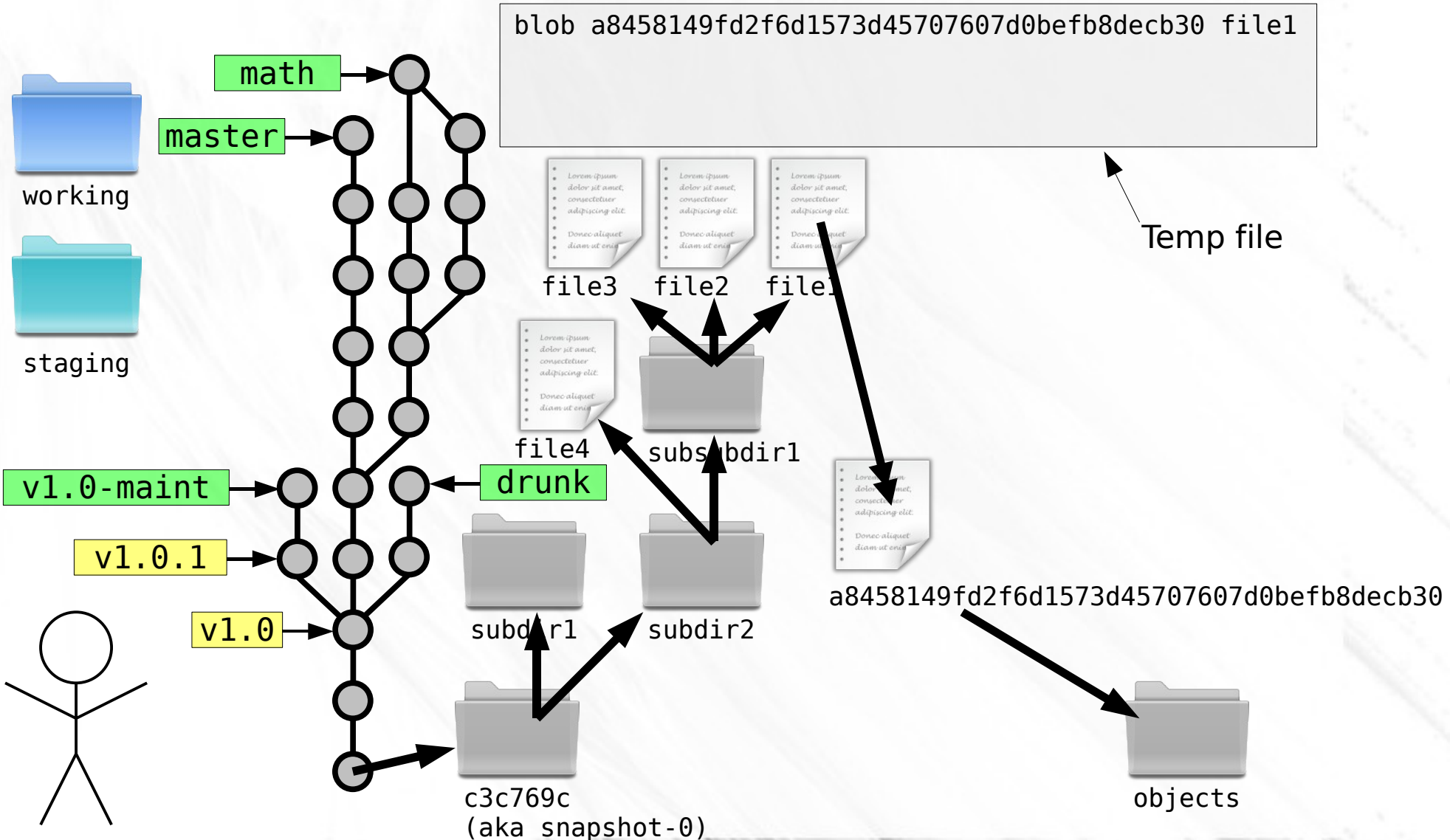
# Eliminating Duplication



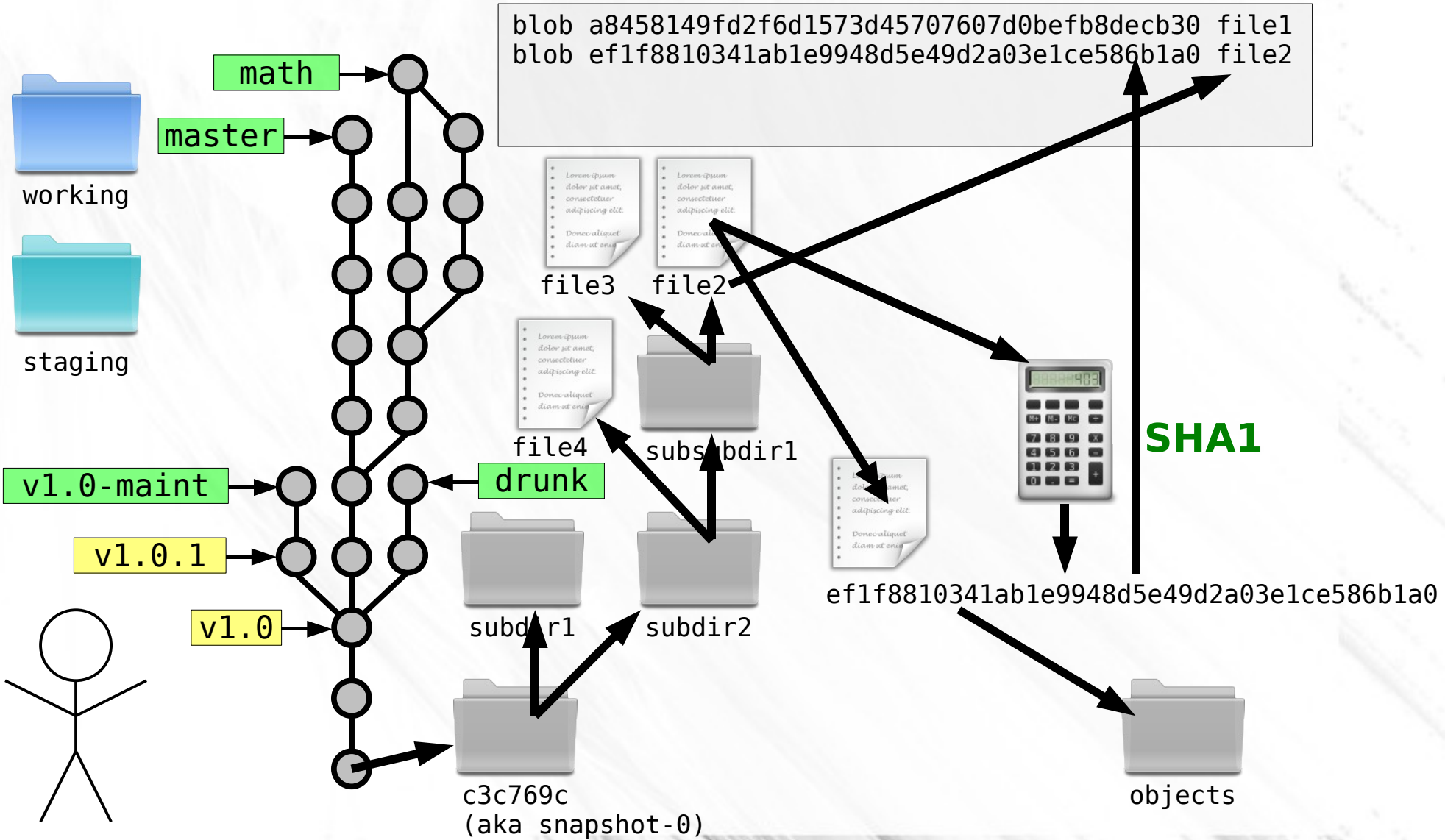
# Eliminating Duplication



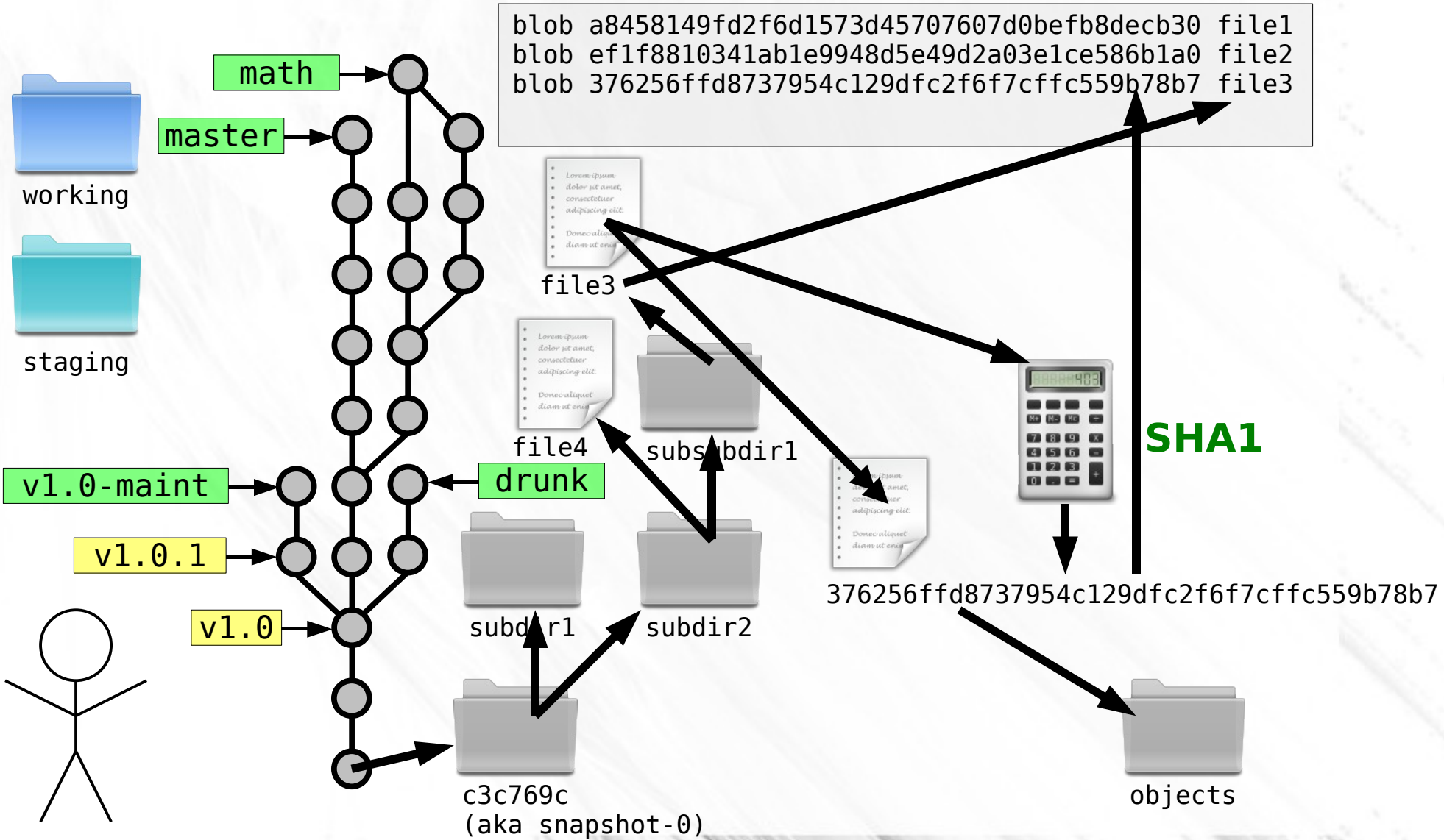
# Eliminating Duplication



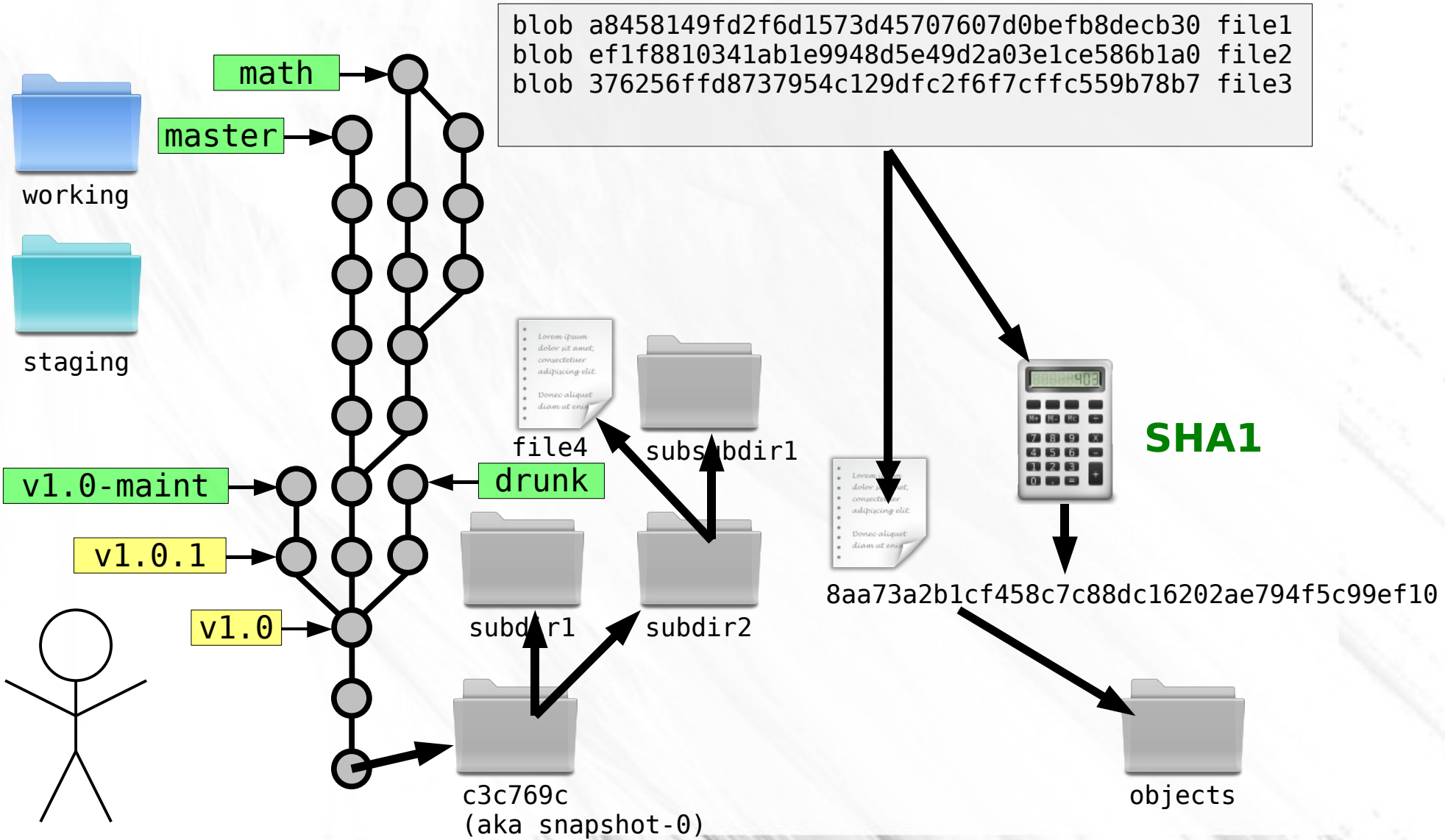
# Eliminating Duplication



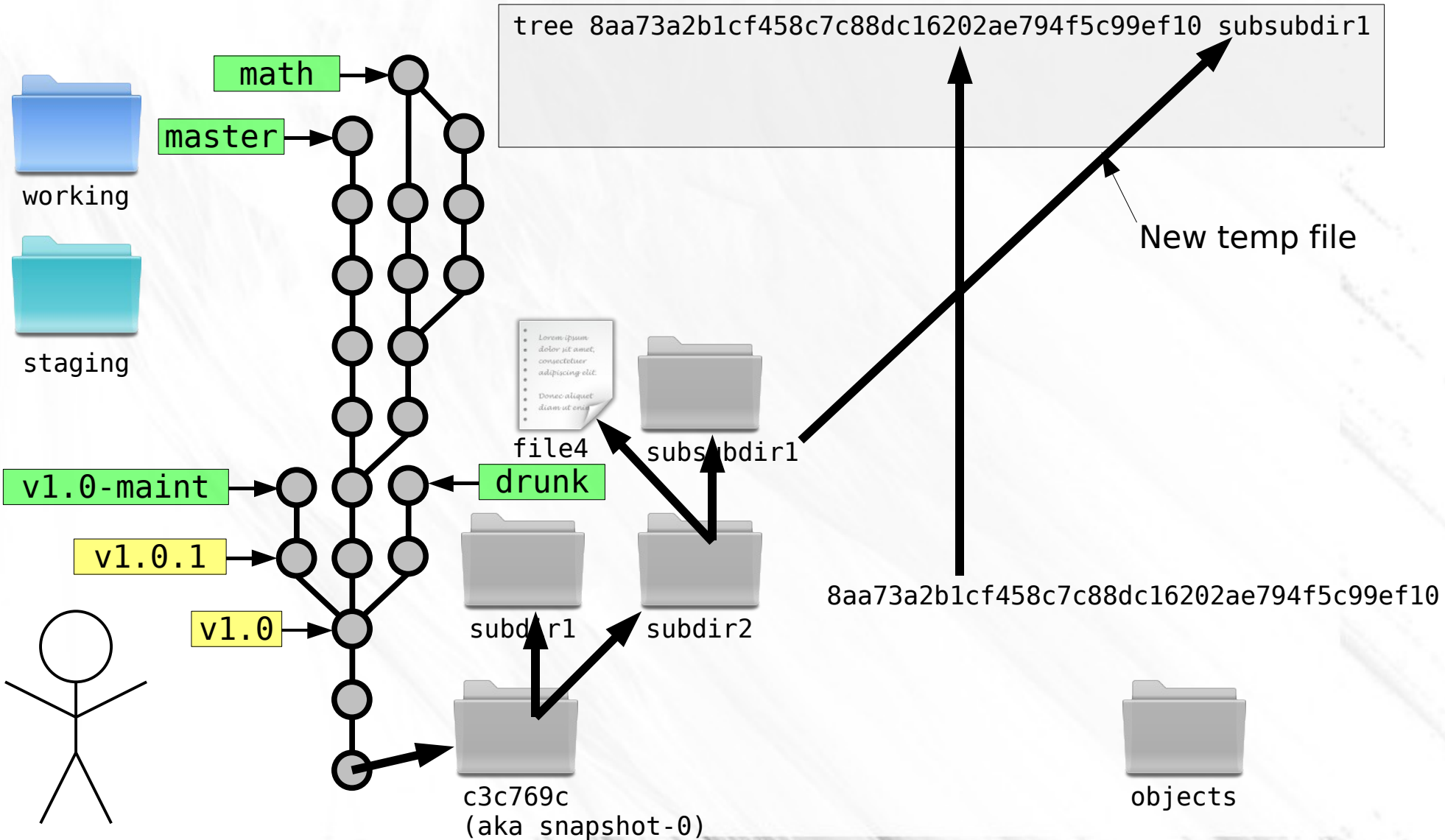
# Eliminating Duplication



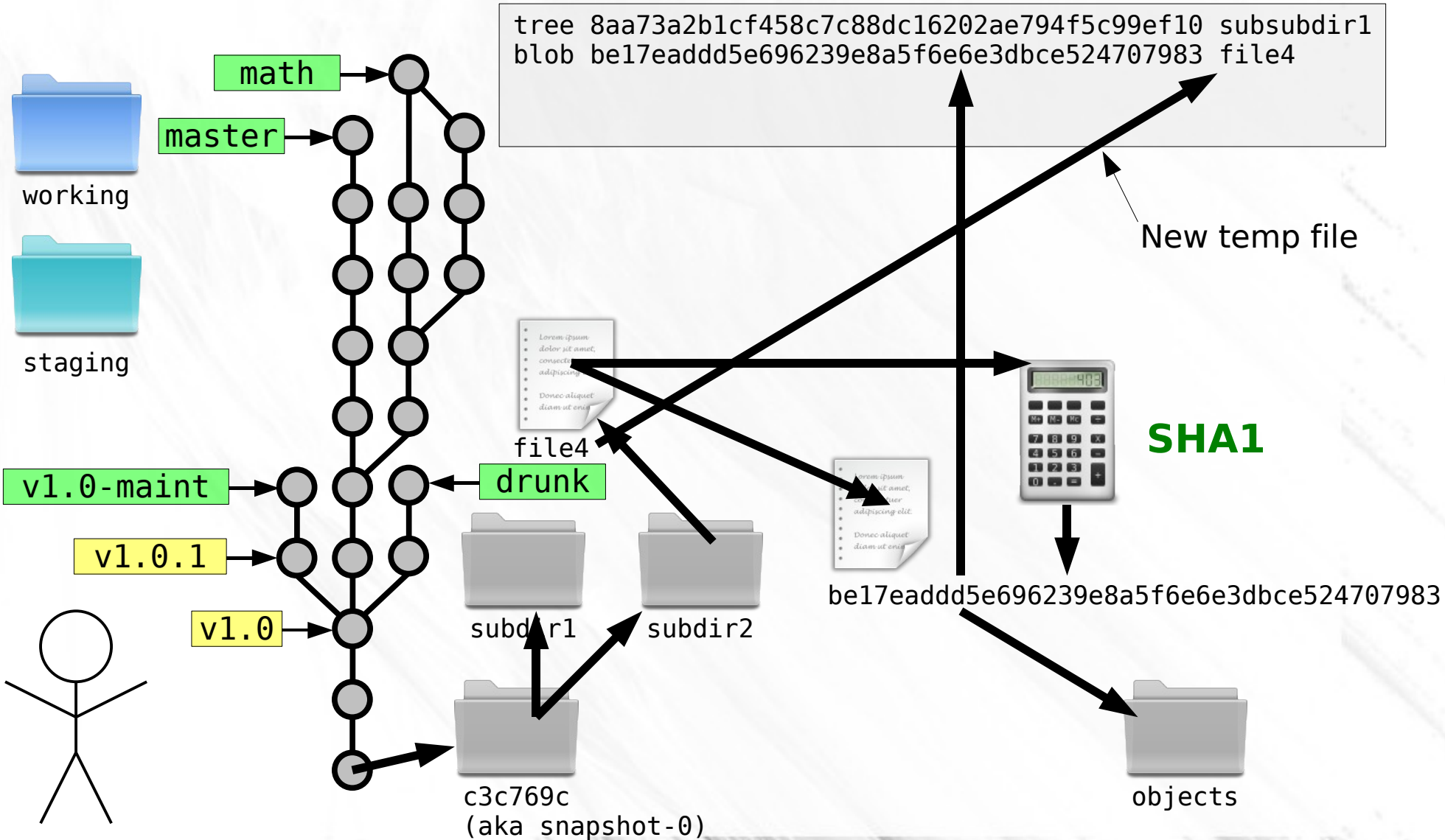
# Eliminating Duplication



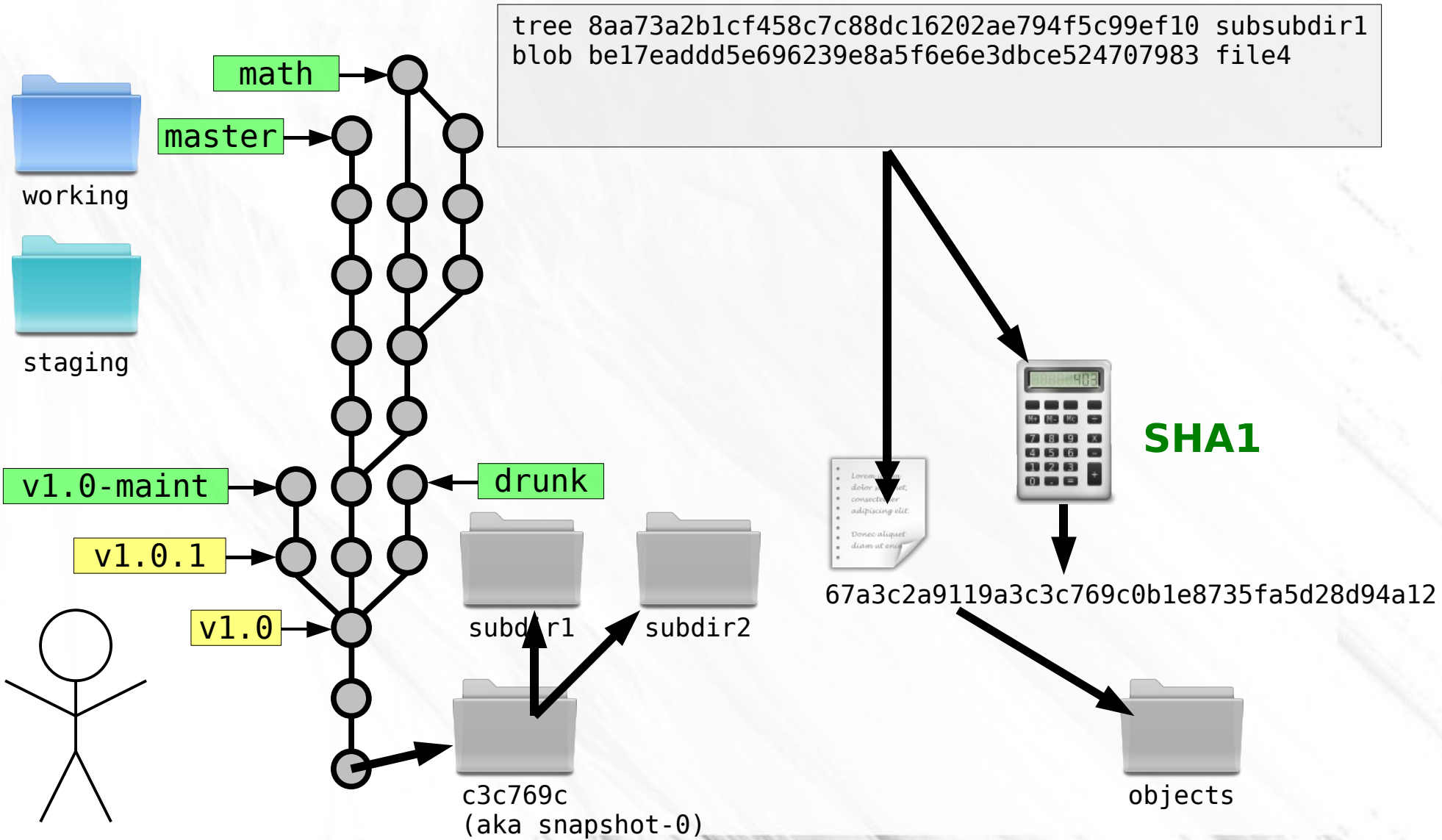
# Eliminating Duplication



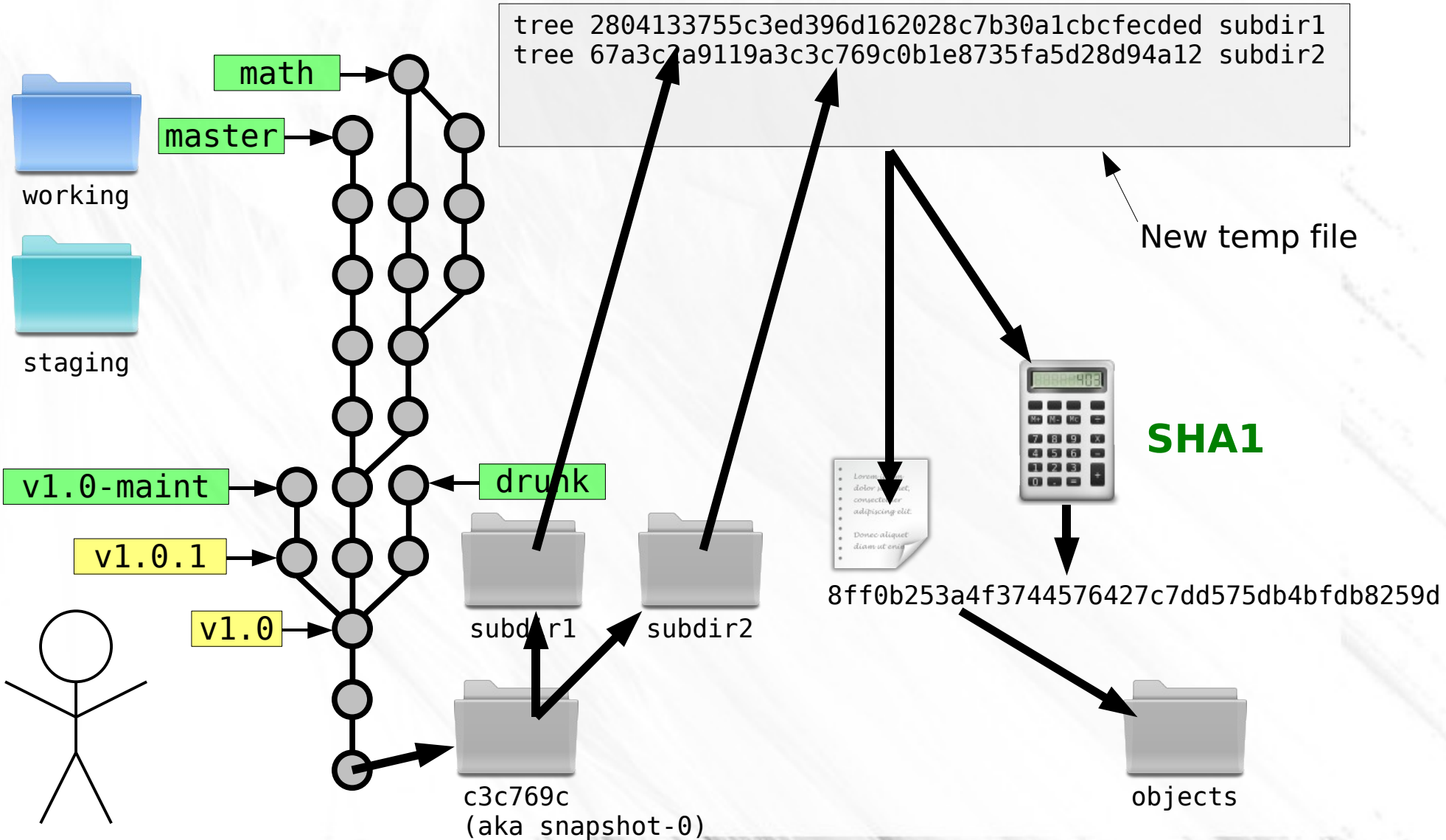
# Eliminating Duplication



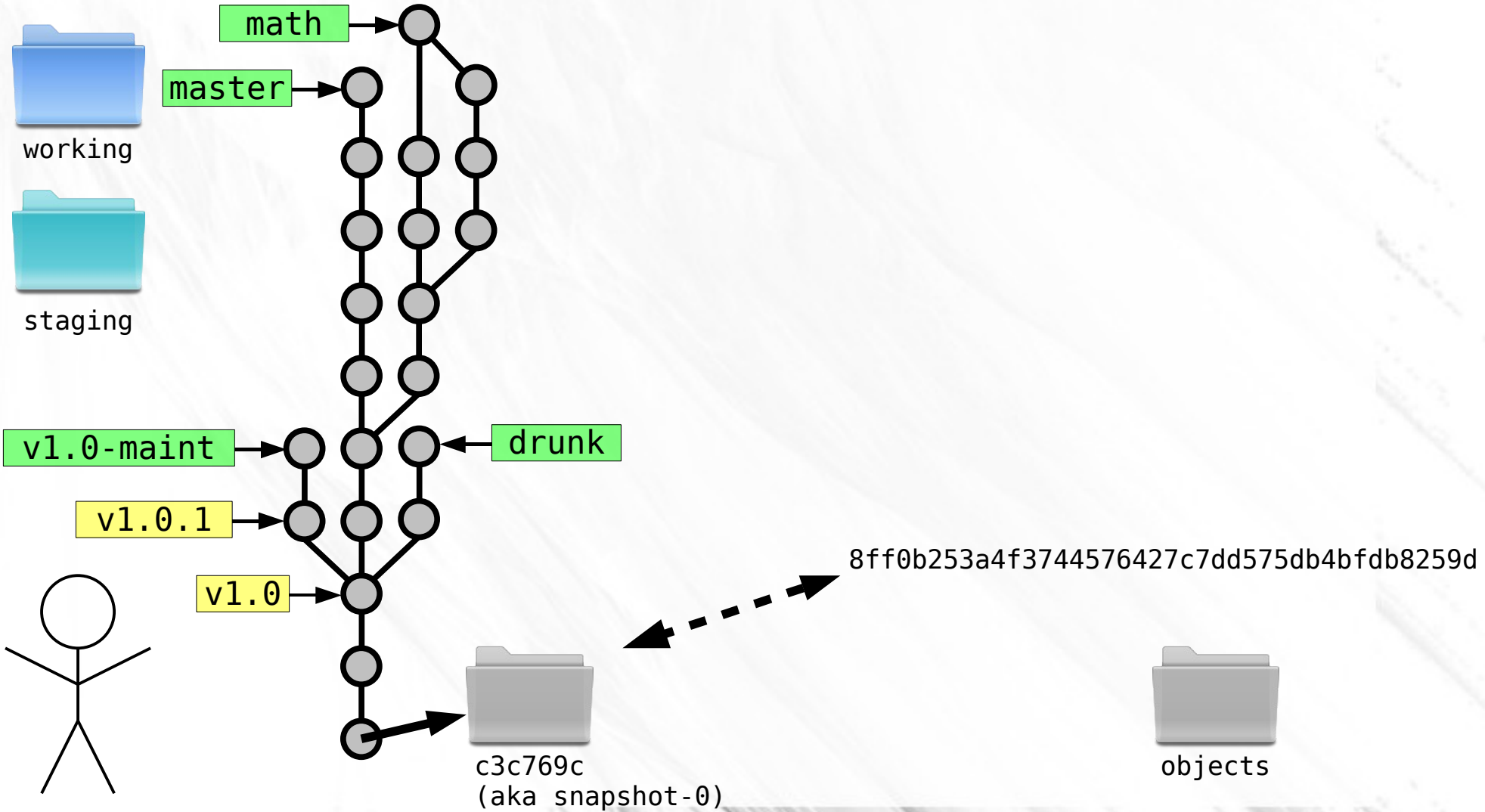
# Eliminating Duplication



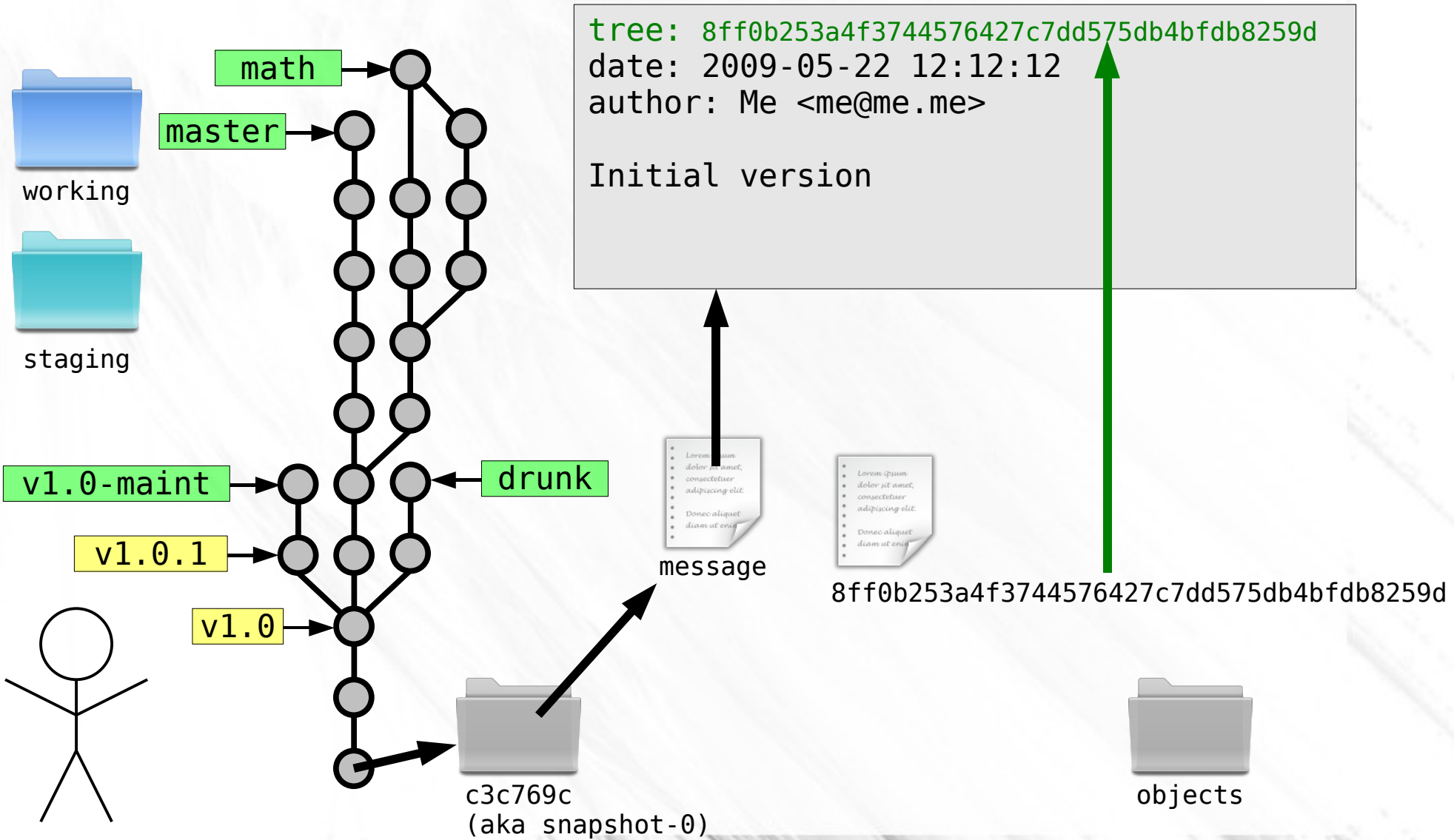
# Eliminating Duplication



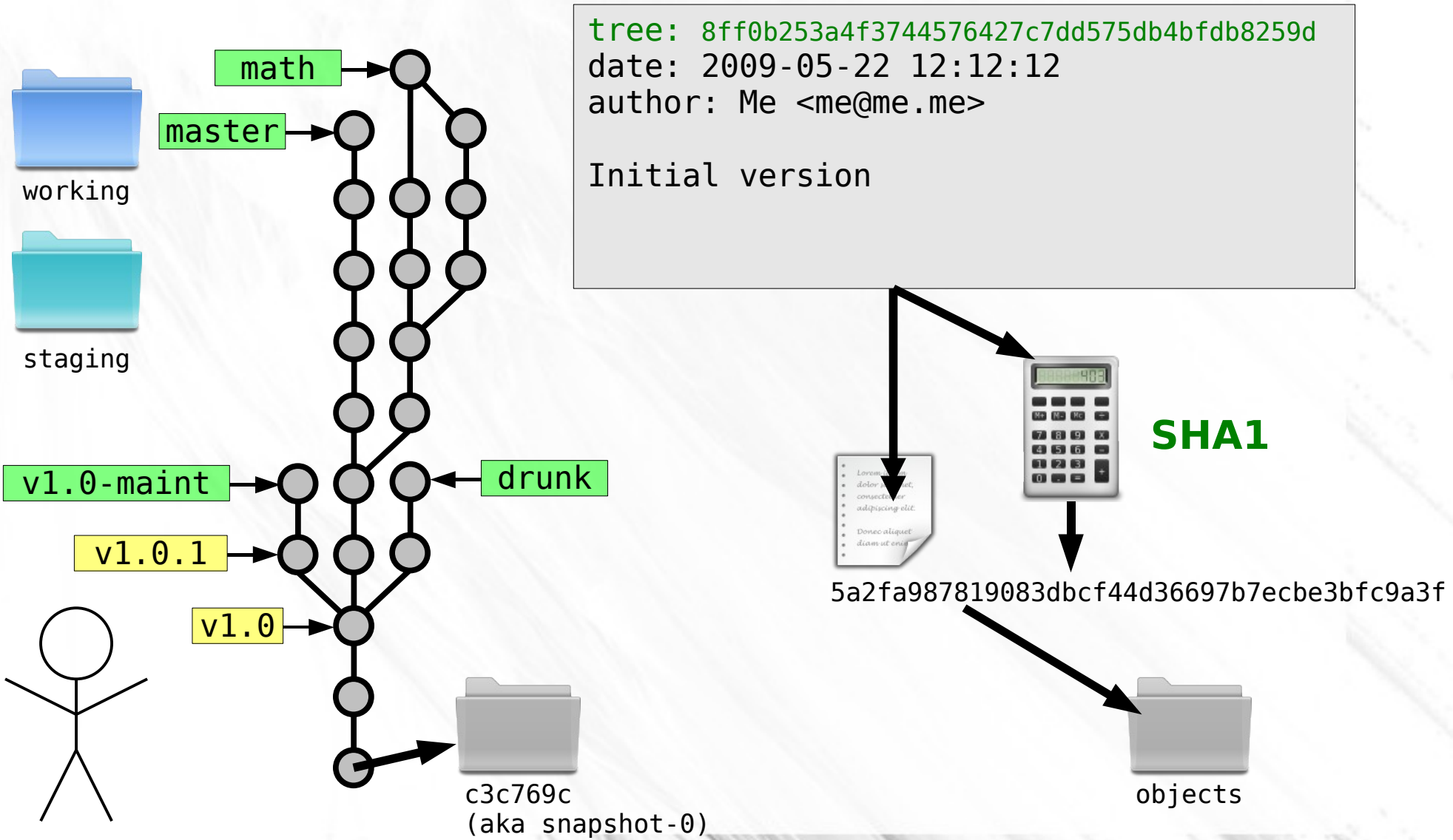
# Eliminating Duplication



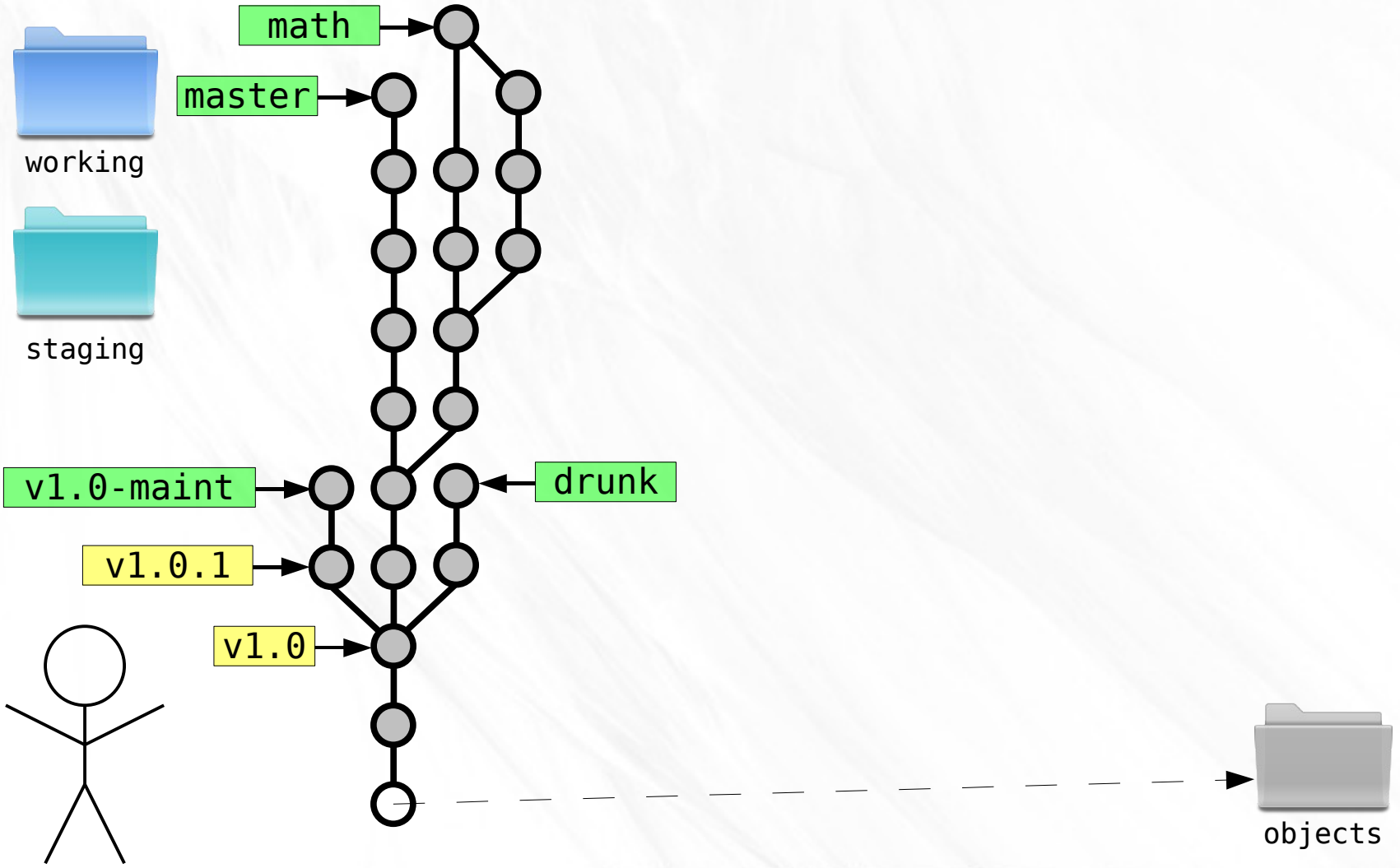
# Eliminating Duplication



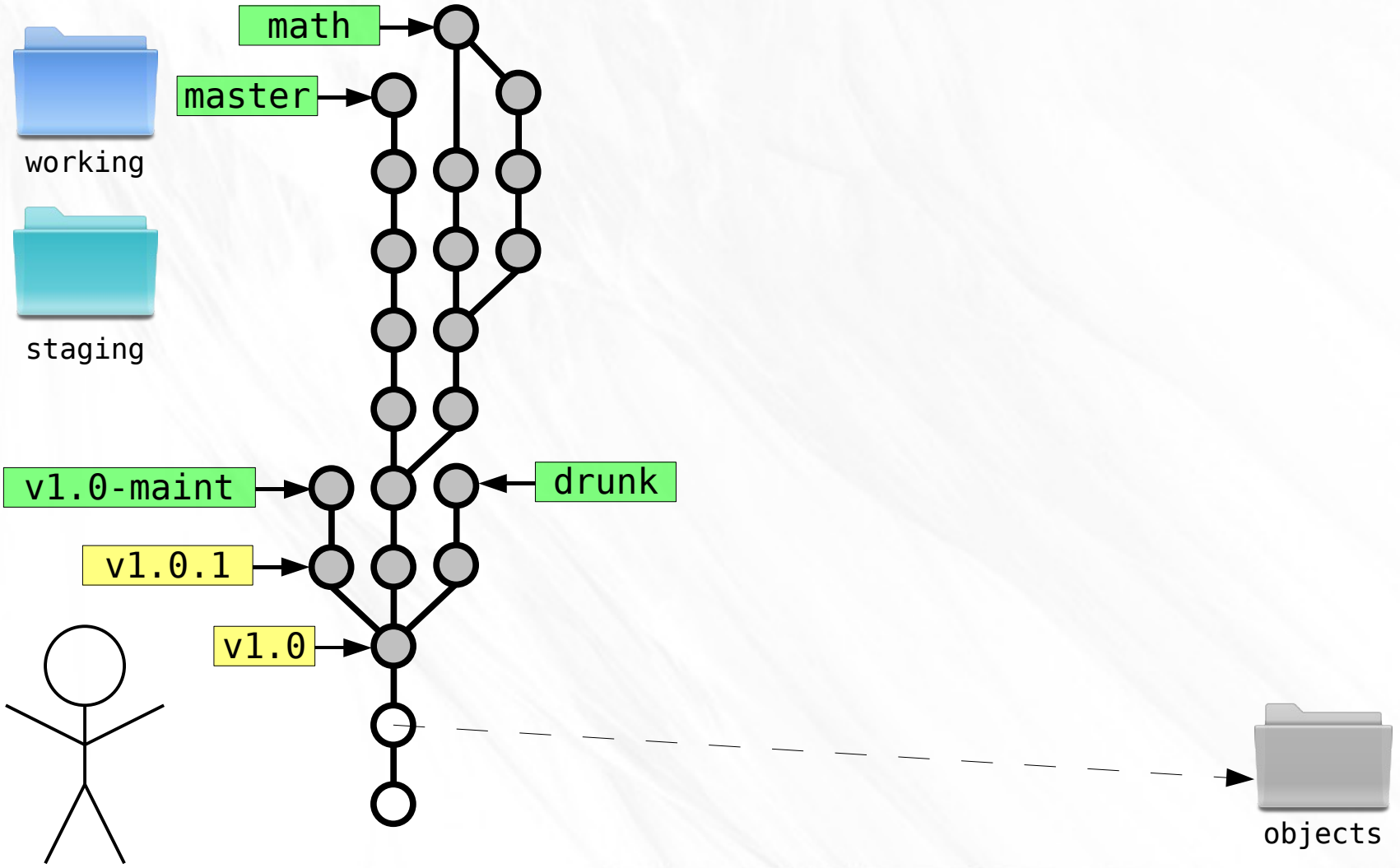
# Eliminating Duplication



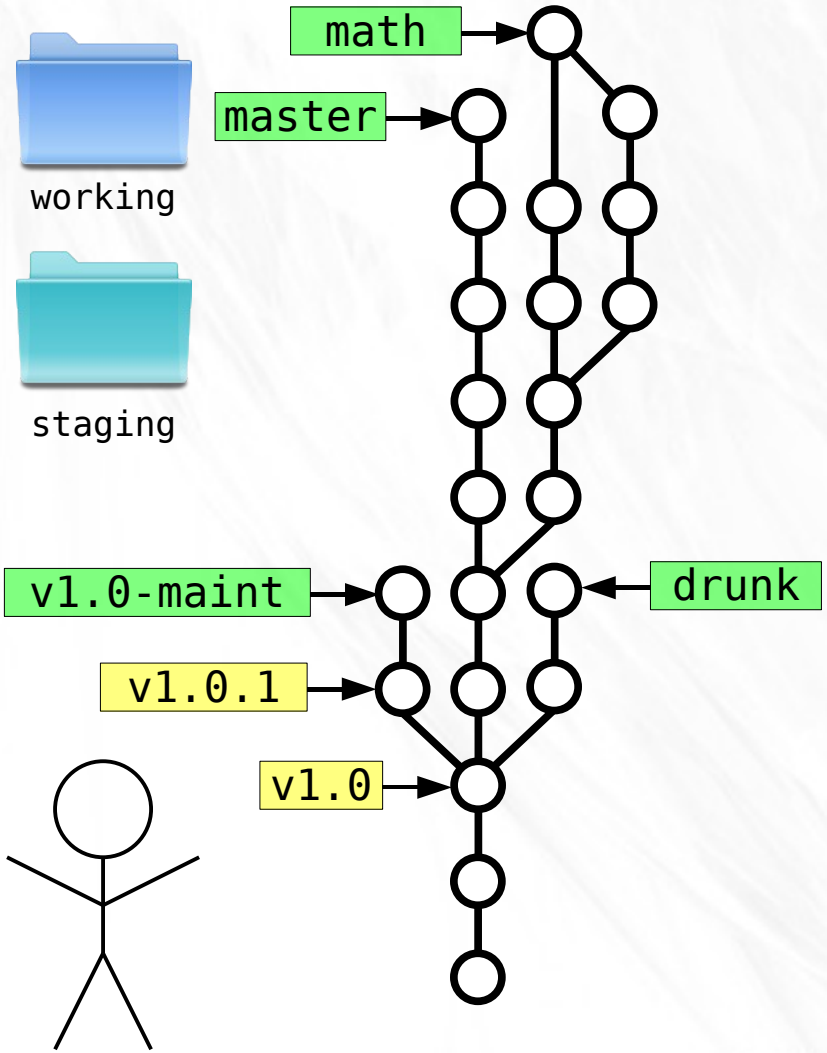
# Eliminating Duplication



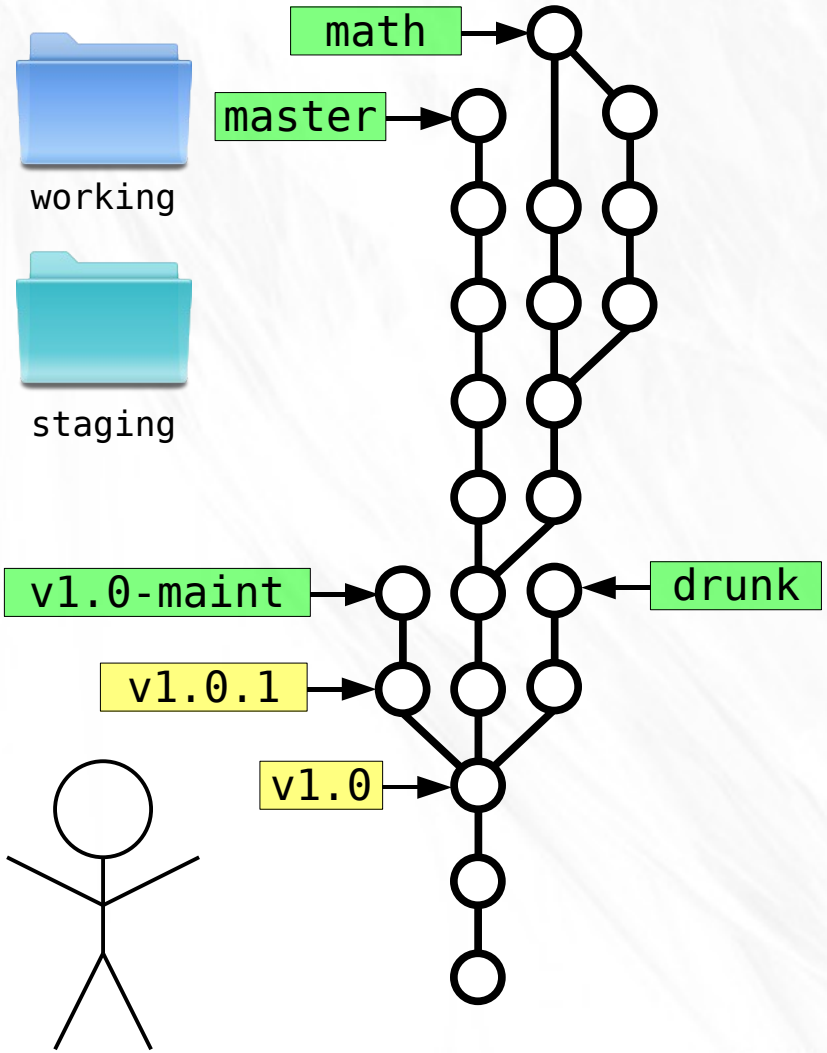
# Eliminating Duplication

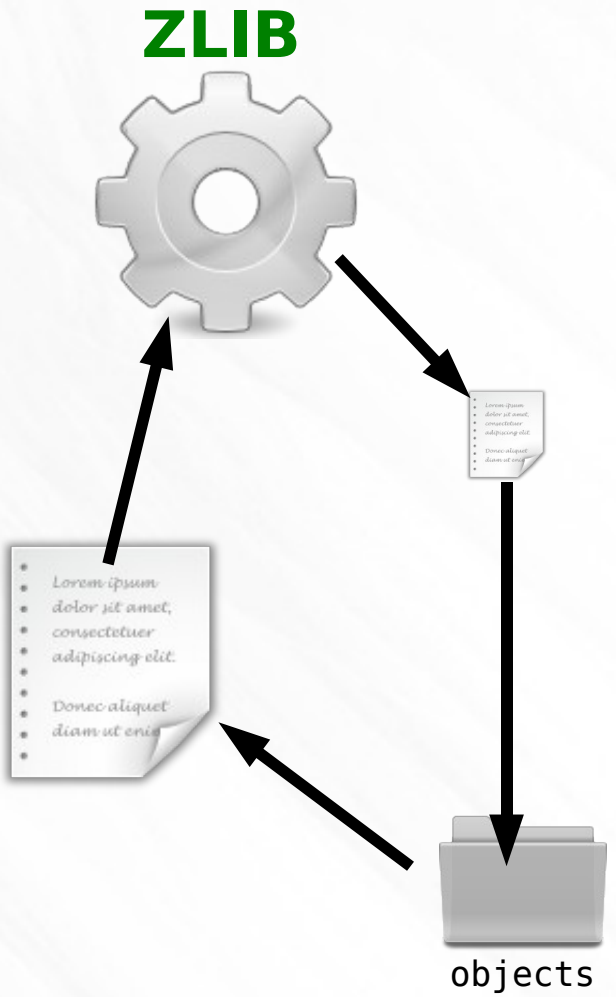


# Eliminating Duplication

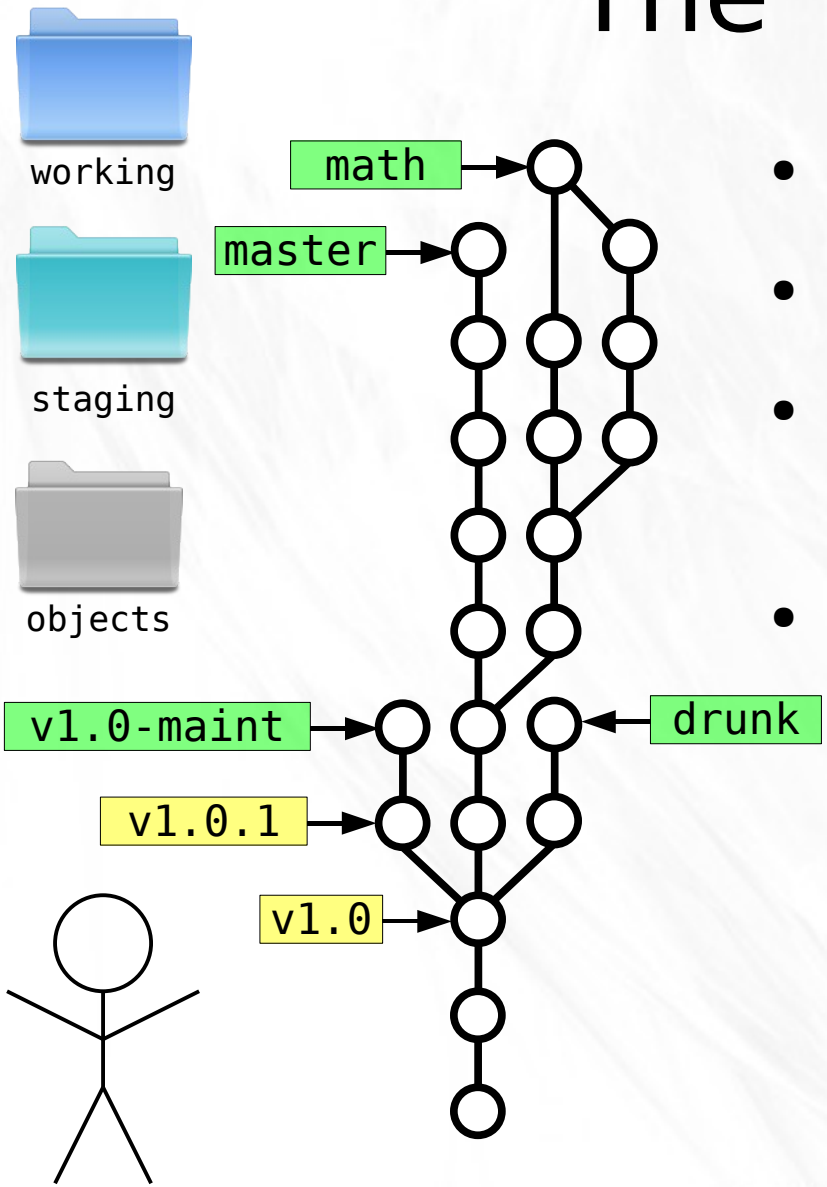



# Compressing Blobs



# The True Git

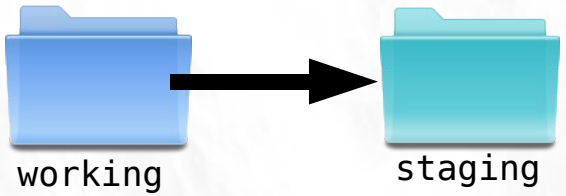



- TADAA!
- This is pretty much Git
- Nicer command line tools for all these operations
- Many, many other tools

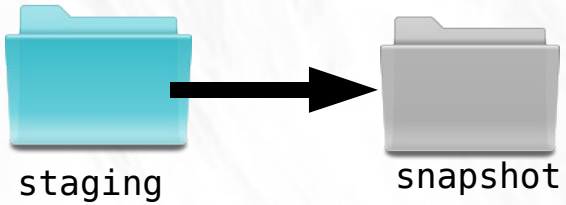
# Commands: Getting Started

- First, tell Git who you are:
  - `git config --global user.name “My Name”`
  - `git config --global user.email “my@email.address”`
- Get help:
  - `git <command> -h`
  - `git help <command>`
- Start a new Git repository:
  - `git init`

# Commands: Making snapshots



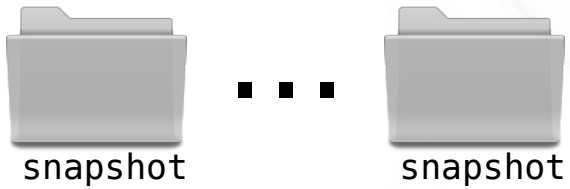

- git add




- git commit

} git commit -a


- git show <snapshot>




- git log – gitk

# Commands: Diffing


vs.


- `git diff`


vs.


- `git diff --staged`


vs.


- `git diff HEAD`


vs.


- `git diff <from> <to>`

# Commands: Branches & Tags

- `git branch`
- `git branch <branch>`
- `git checkout <branch>`

} `git checkout -b ...`

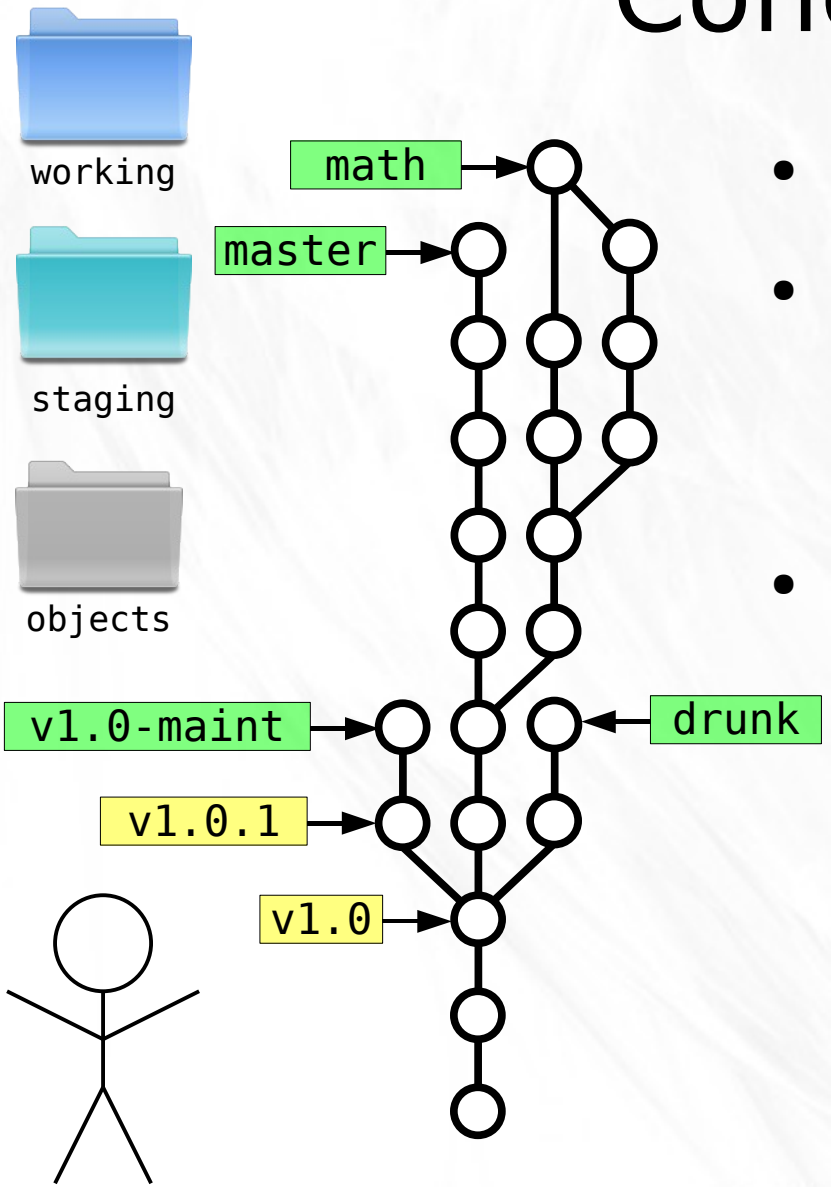
- `git tag -l`
- `git tag <tag>`

# Commands: Fetching & Merging

- `git remote add <name> <URL>`
- `git fetch <name>`
- `git merge <name>/<branch>`

} `git pull`

# Conclusion




- Keep this parable in mind
- Git is simple and powerful

- One more thing:

```
git reflog
```

# Where to go next?

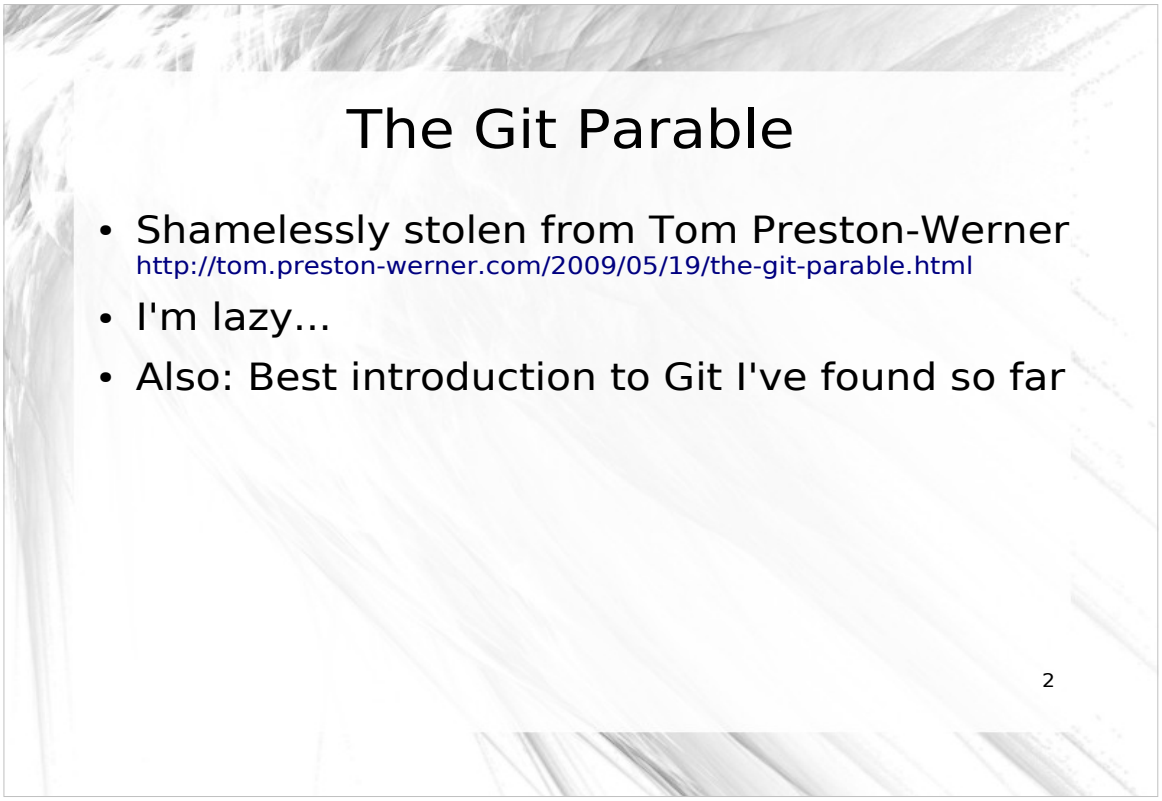
- @ Opera:
  - http://dvcs.oslo.osa
  - dvcs-interest@opera.com
  - johanh@opera.com
- In general:
  - Git homepage: http://git-scm.com
  - Git Community Book: http://book.git-scm.com/
  - http://gitorious.org or http://github.com

# Questions?

- Thanks for your attention!
- This presentation is available at:
  http://dvcs.oslo.osa/~johanh/git_parable/
- Reach me at <johanh@opera.com>


Hi,

My name is Johan Herland. I'm a Core developer in the Oslo office. I have worked in Opera for almost 5 years. I have meddled with distributed version control for about 2 years. I'm also involved in migrating Opera from CVS to Git.




The following story is shamelessly stolen from Tom Preston-Werner. He has written this story, and my only contribution is adapting it to this presentation format.

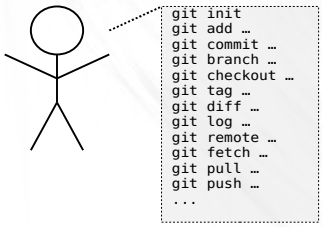
I'm doing this because I'm lazy.

But also because this is the best introduction to Git I have found so far.


Git is a simple and powerful system.

# The Git Parable

- Git - simple & powerful

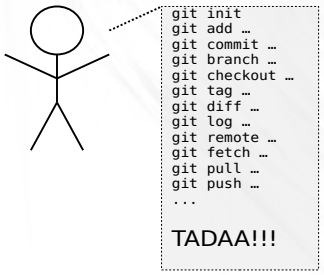





Often, people try to teach Git by demonstrating a few dozen commands, and then yelling...

# The Git Parable

- Git - simple & powerful

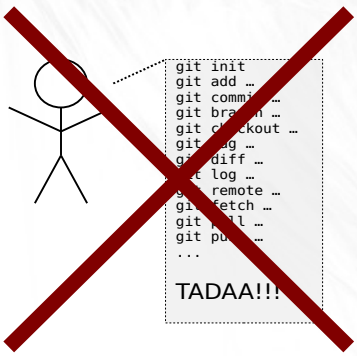





TADAA!!!

## The Git Parable

- Git - simple & powerful






I don't believe this is the best way of teaching Git.

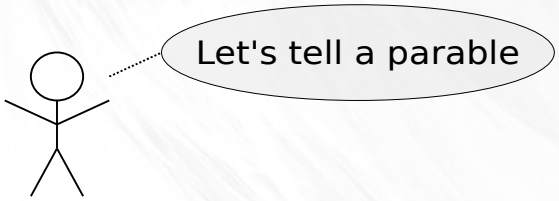
Sure, it will let you use Git to perform simple tasks, but the Git commands will still feel like magical incantations, and doing anything out of the ordinary will be terrifying.

Until you understand the concepts upon which Git is built, you'll feel like a stranger in a foreign land.

Instead, let's tell a story...

# The Git Parable

- Git - simple & powerful

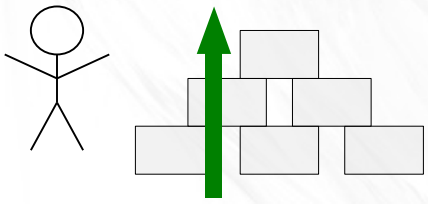





In fact, let's tell a parable.

## The Git Parable

- Git - simple & powerful

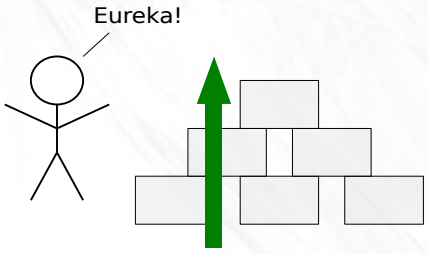


8

The following parable will take you on a journey through the creation of a Git-like system from the ground up.

## The Git Parable

- Git - simple & powerful




9

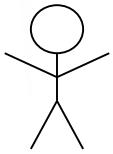
Understanding the concepts will be the most valuable thing you can do to fully grok Git.

The concepts themselves are simple, but allow for an amazing wealth of functionality to spring into existence.

After this parable, you have everything you need to easily master the various Git commands, and become a Git power user.

## The Parable

- A simple computer
  - A text editor
  - A few filesystem commands

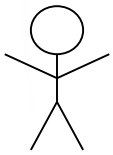




Imagine that you have a simple computer with absolutely nothing but a text editor, and some simple filesystem commands.

## The Parable

- Write a large software program

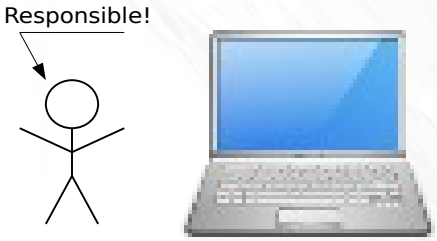


11

Now, imagine that you have decided to write a large software program on this system.

## The Parable

- Write a large software program
- Invent some method to keep track of versions
  - retrieve code that you changed/deleted



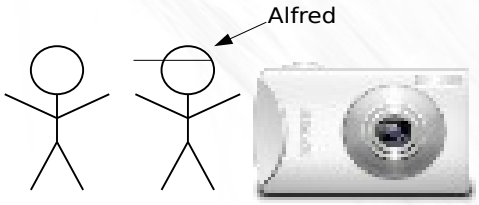

12

Because you are a responsible software developer, you decide that you need to invent some method to keep track of versions of your software, so that you can retrieve code that you previously changed or deleted.

What follows is a story about how you might design one such version control system, and the resoning behind those design choices.

# Snapshots

- Alfred, the photographer

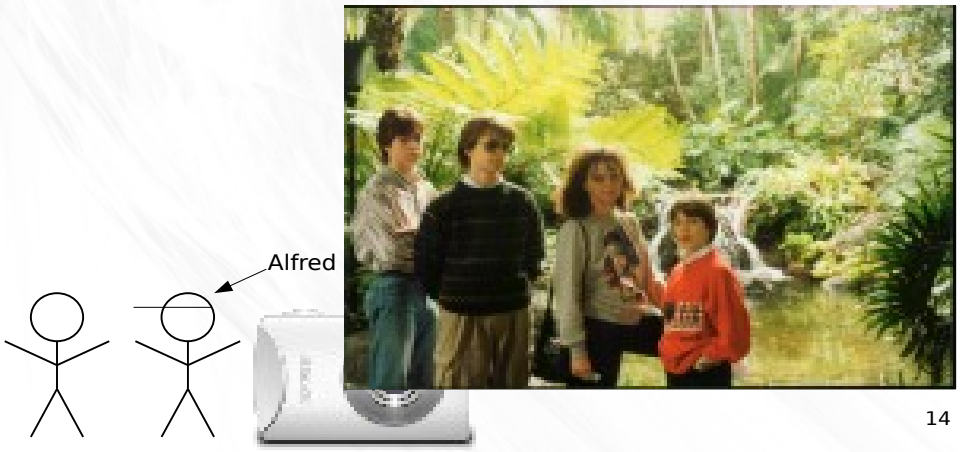





Alfred is a friend of yours that works as a photographer in one of those “Special Moments” photo boutiques.

# Snapshots

- Alfred, the photographer

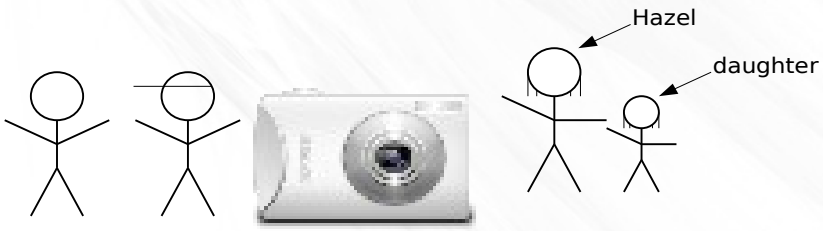





All day long he takes photos of little kids posing awkwardly in front of some tacky jungle backdrop.

# Snapshots

- Alfred, the photographer
- Hazel and her daughter

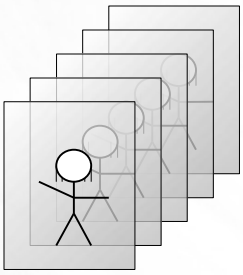


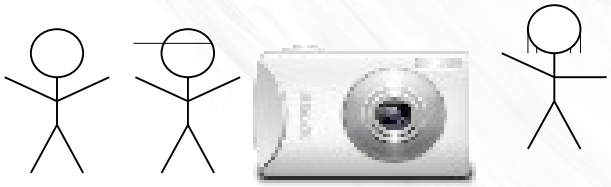



Alfred tells you a story about a woman named Hazel, who brings her daughter in for a portrait every year on the same day.

# Snapshots

- Alfred, the photographer
- Hazel and her daughter
  - Remember what the daughter was like at each different stage



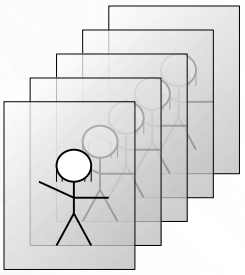




"She likes to remember what her daughter was like at each different stage, as if the snapshots really let her move back and forth in time to those saved memories."

## Snapshots

- Alfred, the photographer
- Hazel and her daughter
  - Remember what the daughter was like at each different stage

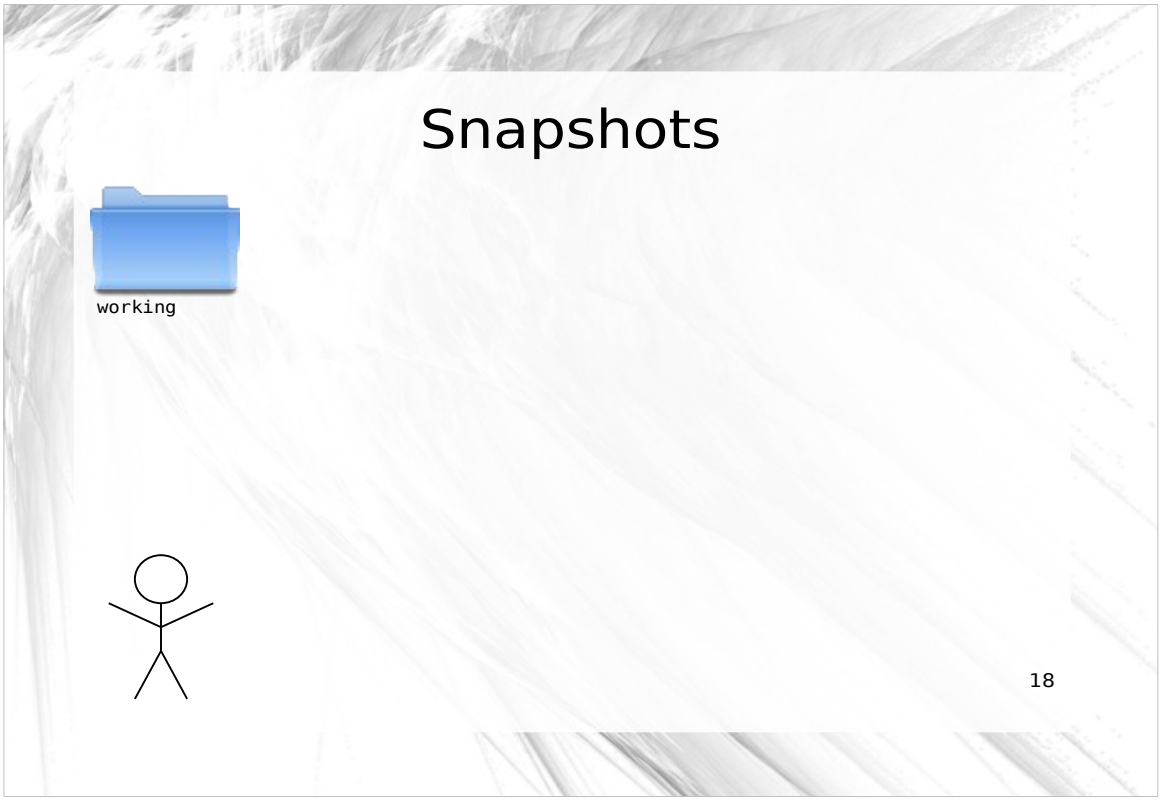





You suddenly see the ideal solution to your version control dilemma: Snapshots, like save point in a video game, are really what you care about in your version control system.

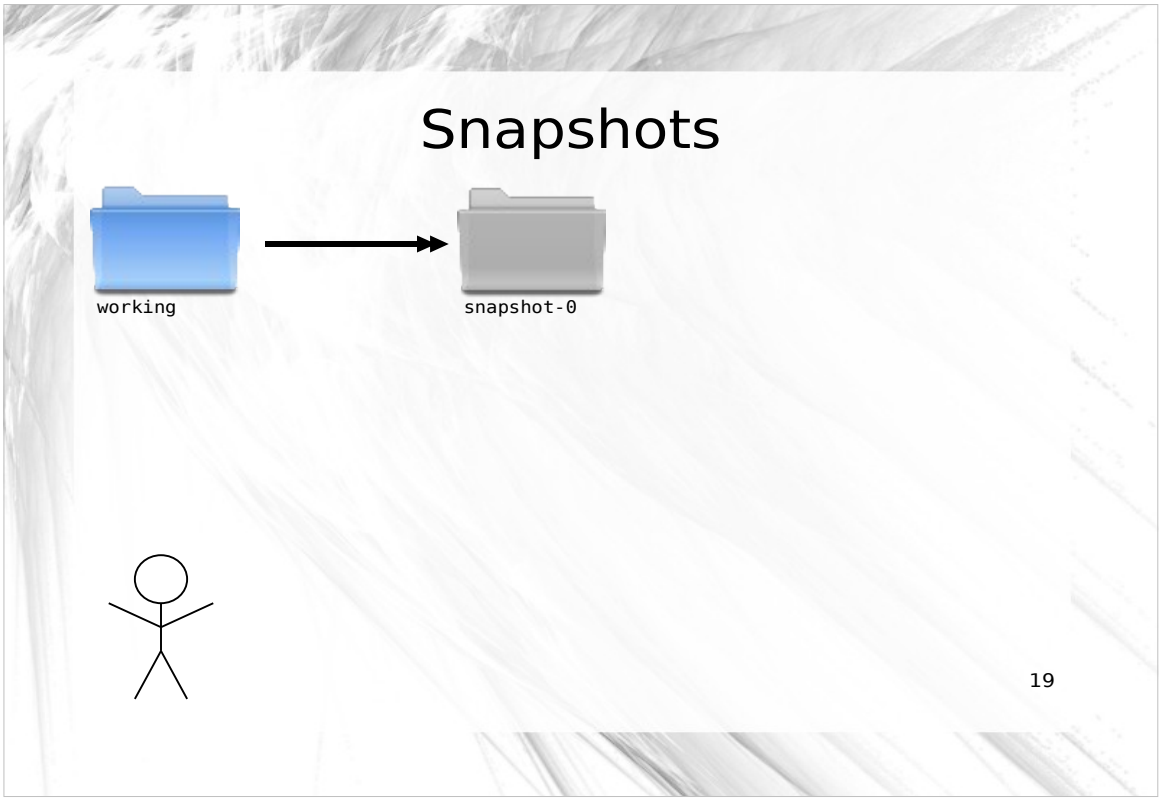
What if you could take snapshots of your codebase at any time, and resurrect that code on demand?

You go back to your computer and start working...



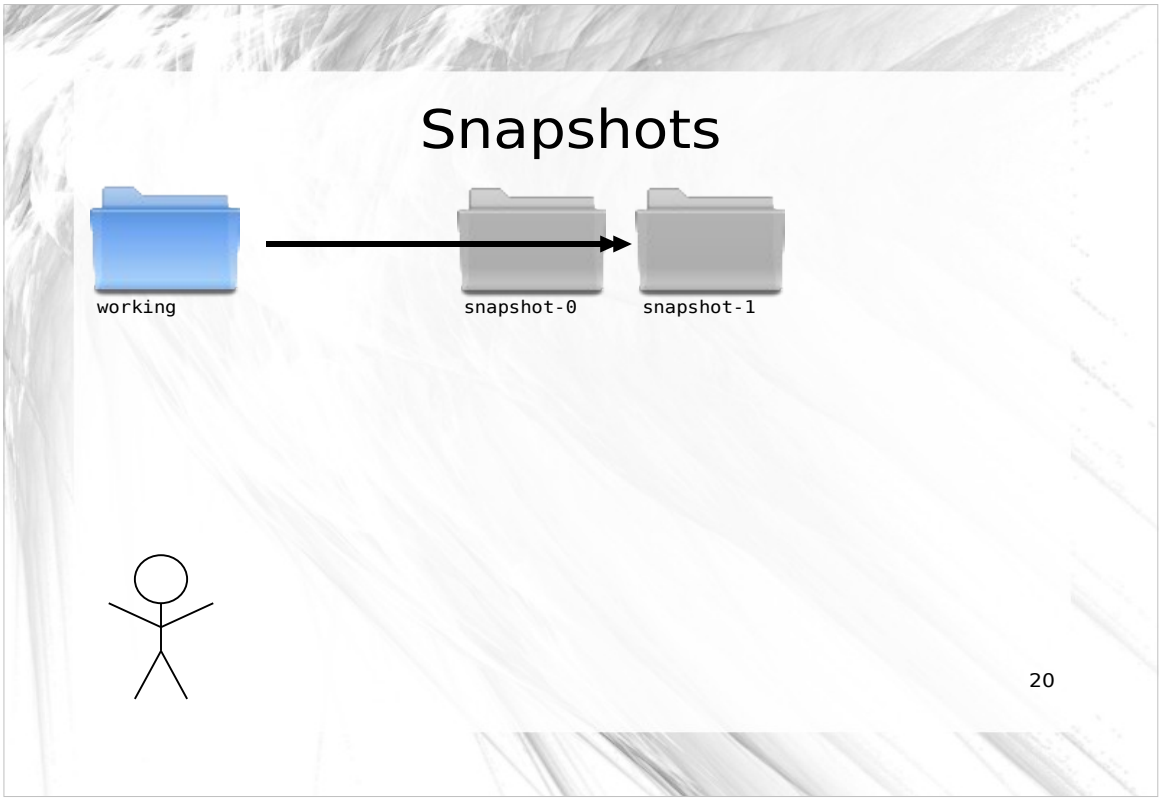

You start your project in a directory named “working”.

As you code, you try to write one feature at a time.

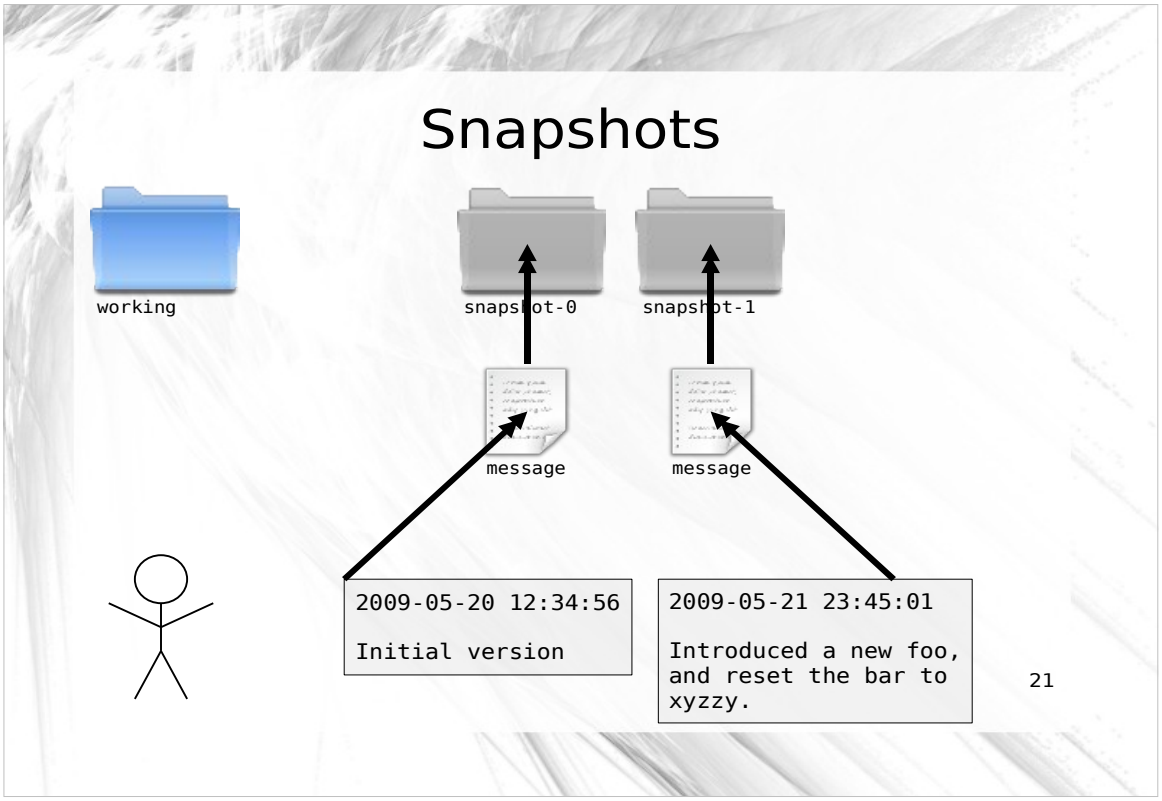



When you complete a self-contained portion of a feature, you make sure that all your files are saved, and then make a copy of the entire working directory, giving it the name “snapshot-0”.

After you make the copy, you make sure to NEVER again change the code in “snapshot-0”.

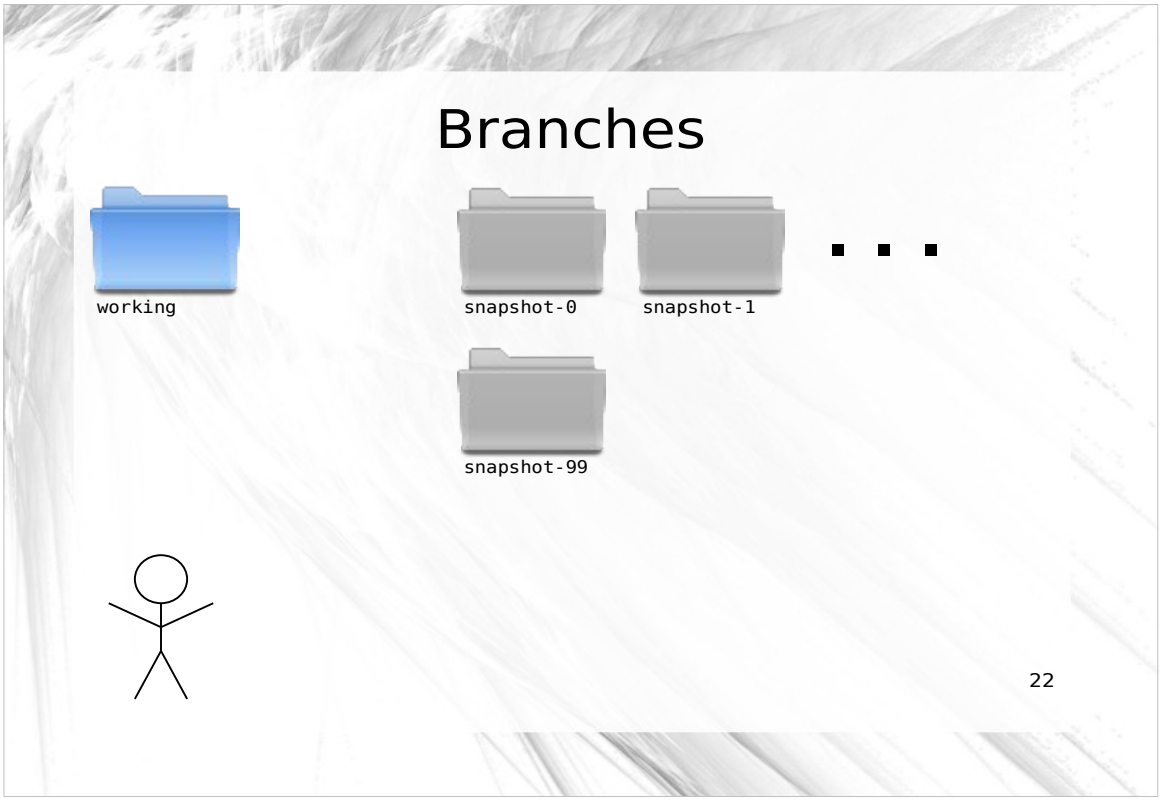



After the next chunk of work, you perform another copy, only this time the new directory gets the name “snapshot-1”, and so on.

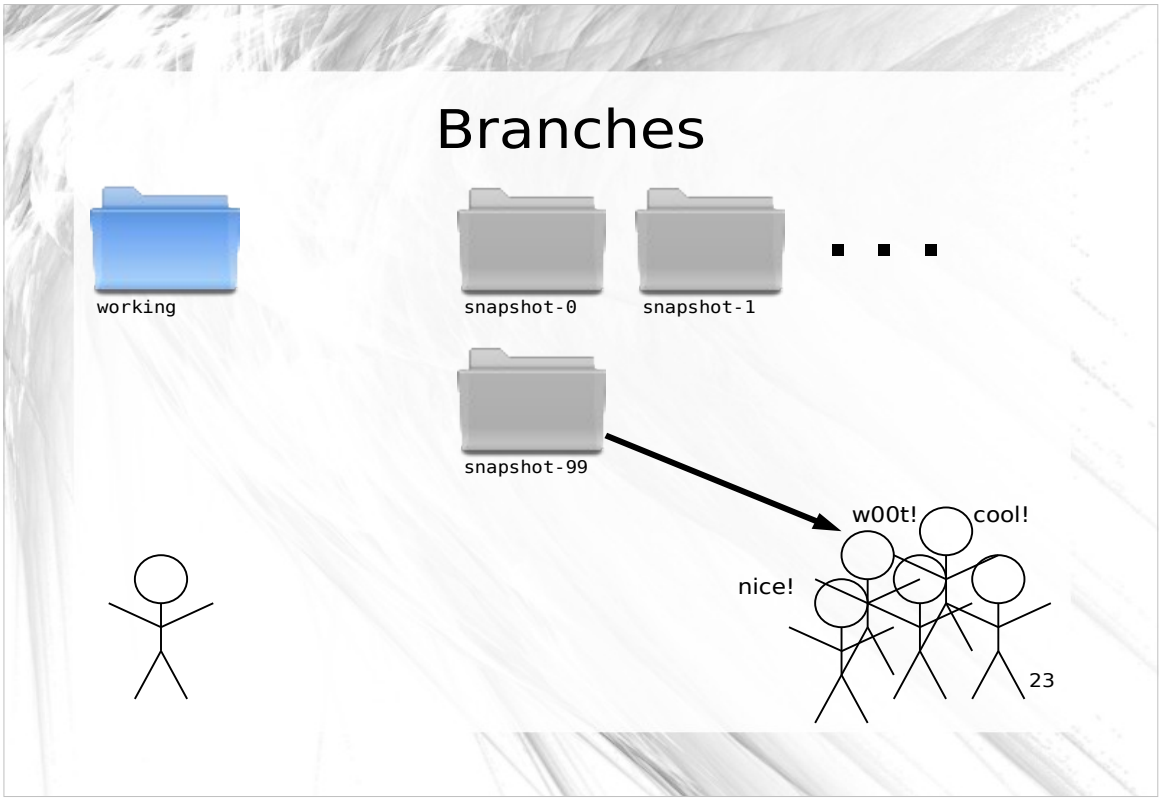



To make it easy to remember what changes you made in each snapshot, you add a special file named “message” to each snapshot directory that contains a summary of the work that you did and the date of completion.

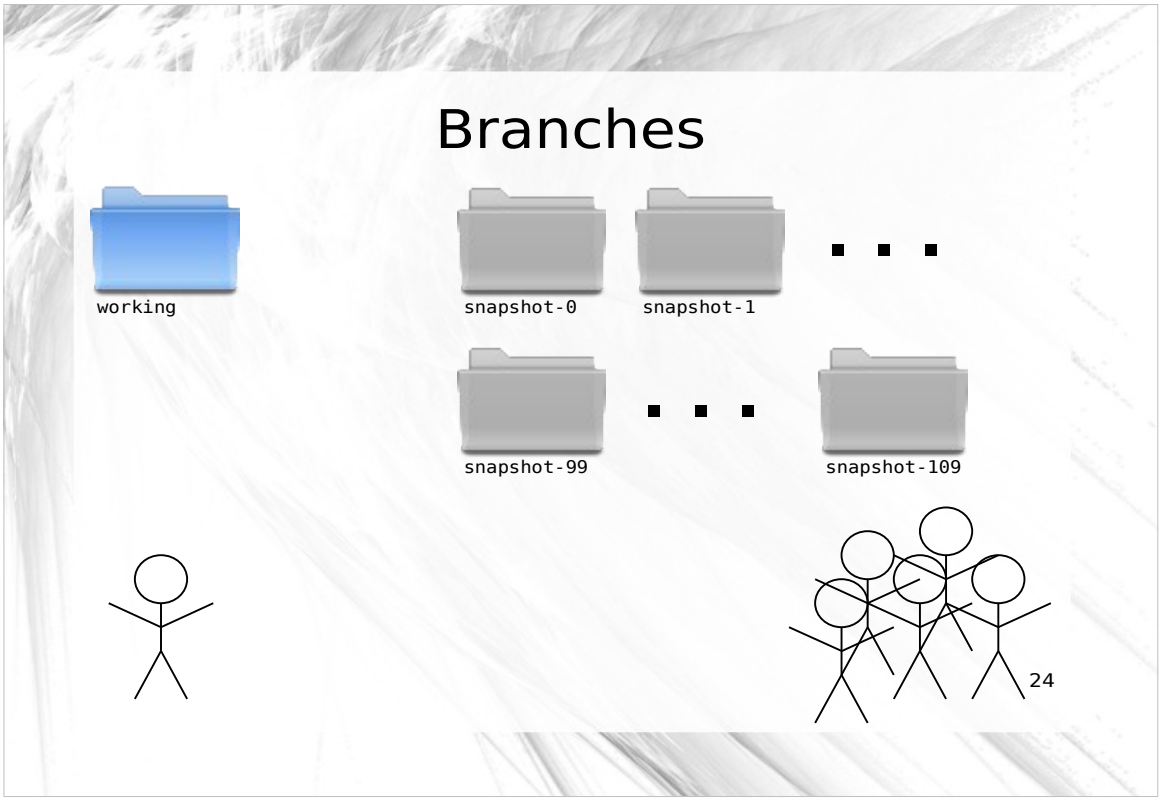
By printing the contents of each message, it becomes easy to find a specific change that you made in the past, in case you need to resurrect some old code.



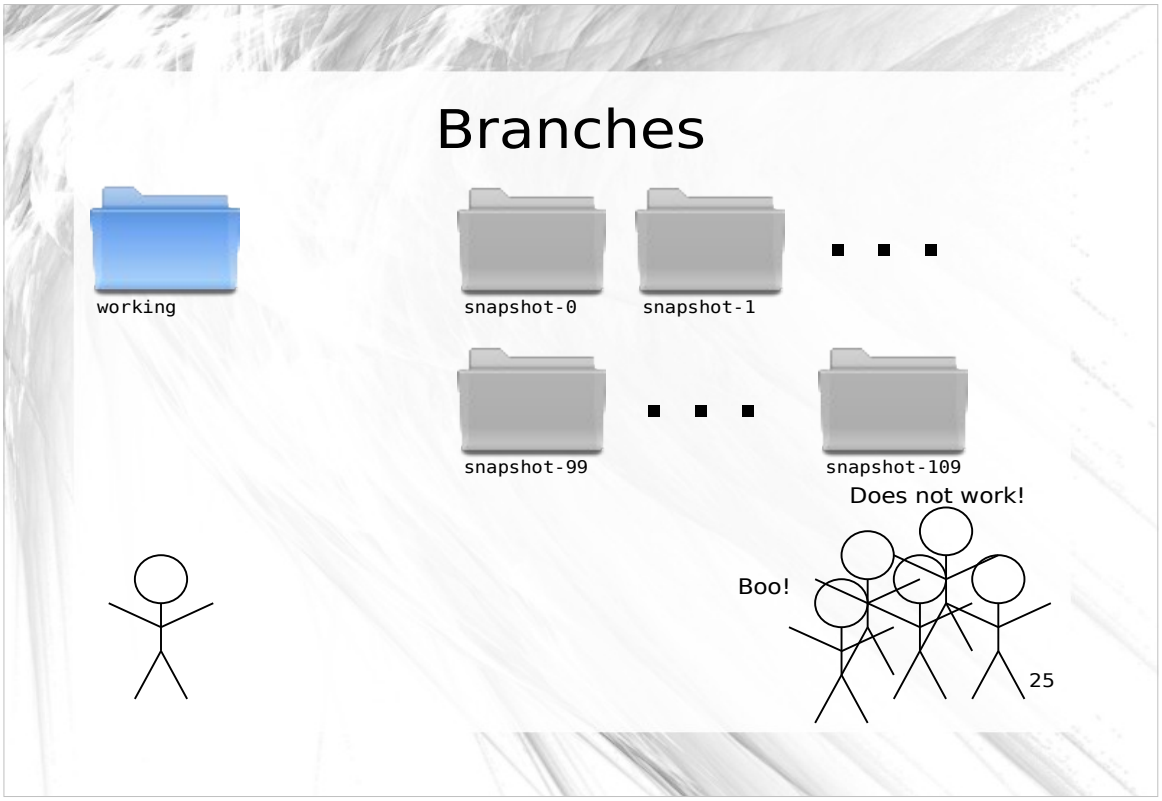

After a bit of time on the project, a release candidate begins to emerge. Late nights at the keyboard finally yield “snapshot-99”. You decide to release this as Version 1.0...




So, “Snapshot-99” is packaged and distributed as Version 1.0 to the eagerly awaiting masses. Stoked by excellent response to your software, you push forward, determined to make the next version an even bigger success.

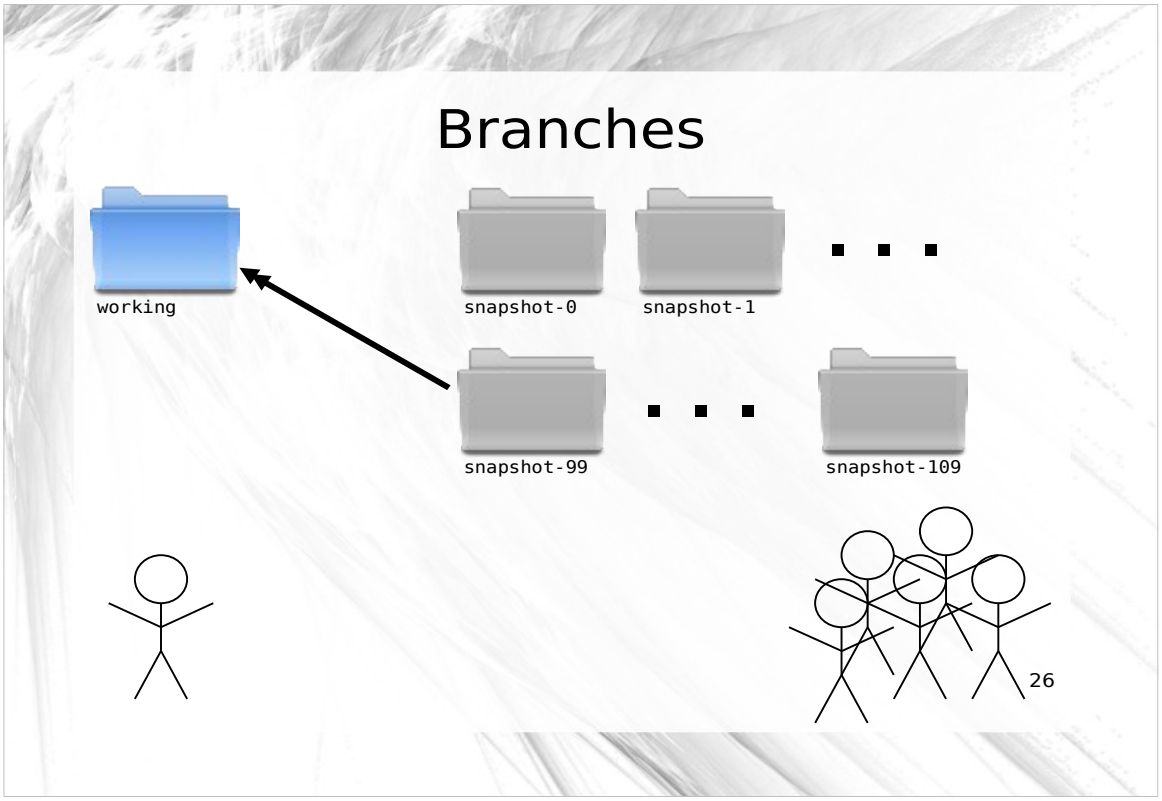



You keep adding new features, and make 10 new snapshots.



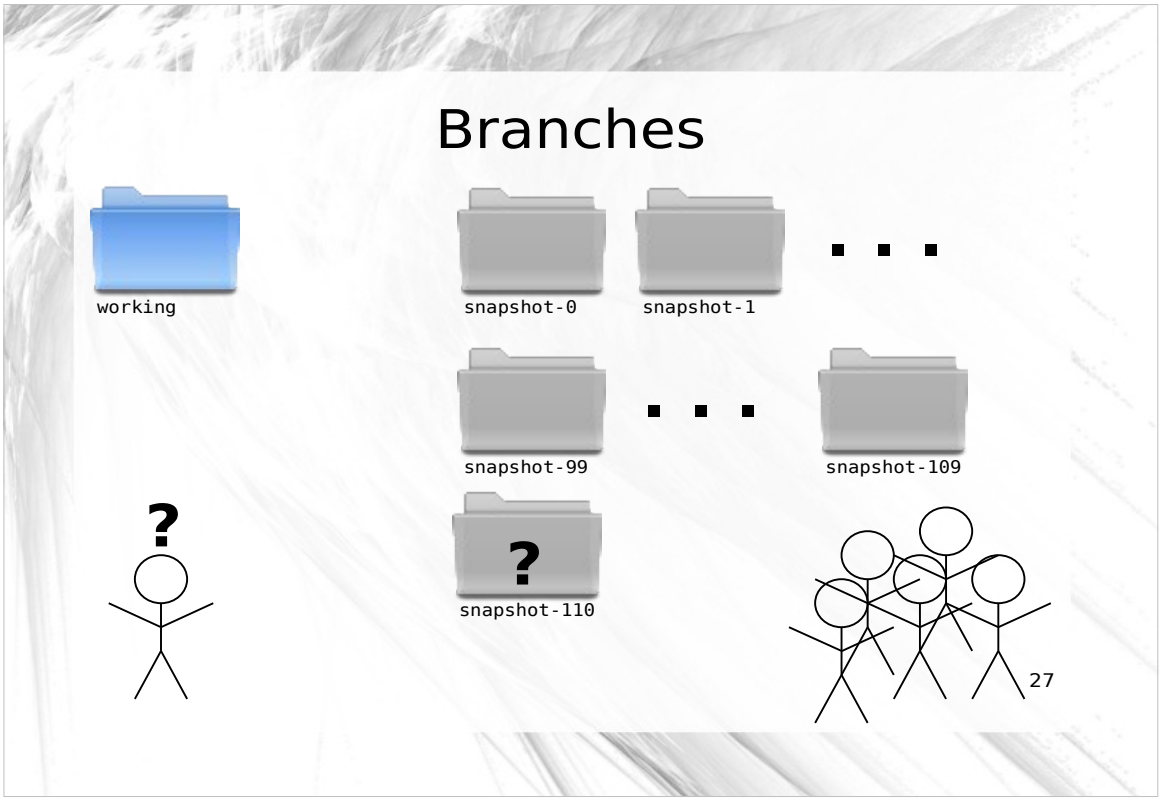

Your VCS has so far been a faithful companion. Old versions of your code are there when you need them and can be accessed with ease.

But not long after the release, bug reports start to come in.




You tell yourself that nobody’s perfect, and copy “snapshot-99” to “working” so that your working directory is at exactly the point where Version 1.0 was released. A few swift lines of code and the bug is fixed in the working directory.

It is here that a problem becomes apparent.




The VCS deals very well with linear development, but for the first time ever, you need to create a new snapshot that is not a direct descendent of the preceding snapshot. If you create a “snapshot-110”, then you’ll be interrupting the linear flow and will have no way of determining the ancestry of any given snapshot.

Clearly, you need something more powerful than a linear system.

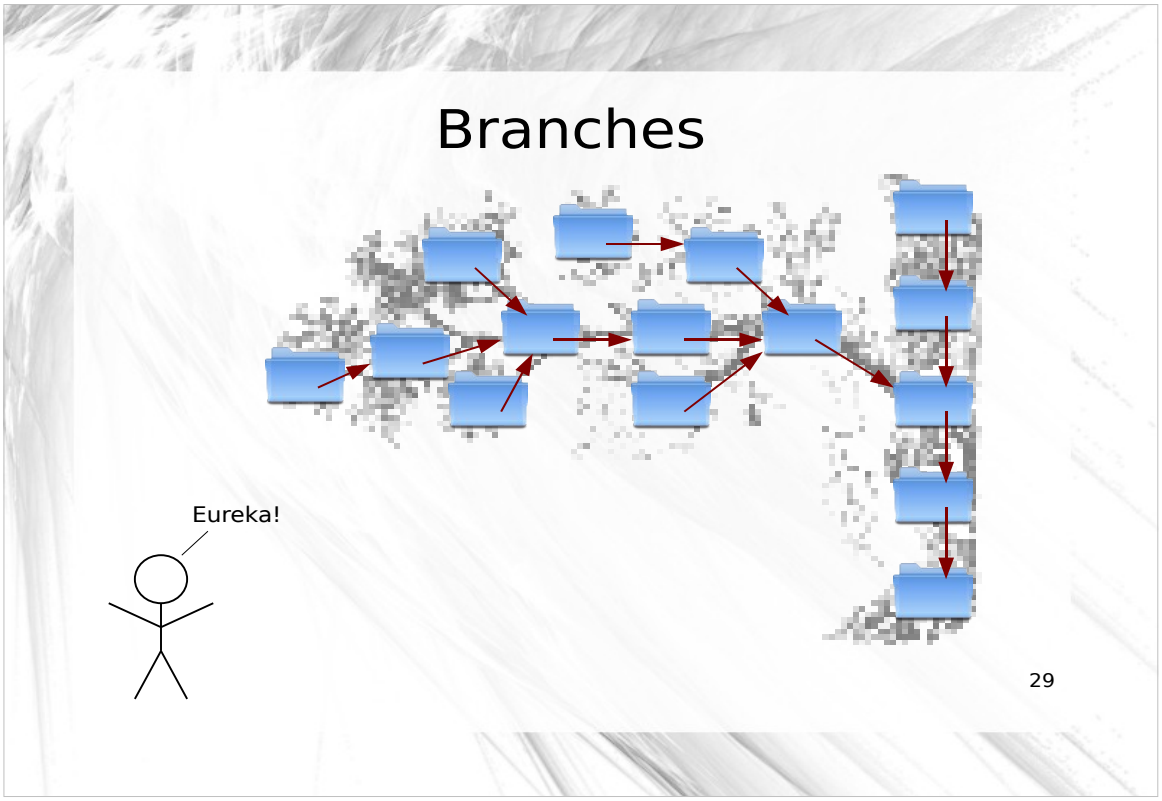
It's time to have a break, and you take a walk outside.


You walk past an oak tree, and start looking more closely at it.

You examine one of the thousands of branch tips, and you idly try to follow it back to the solitary trunk.

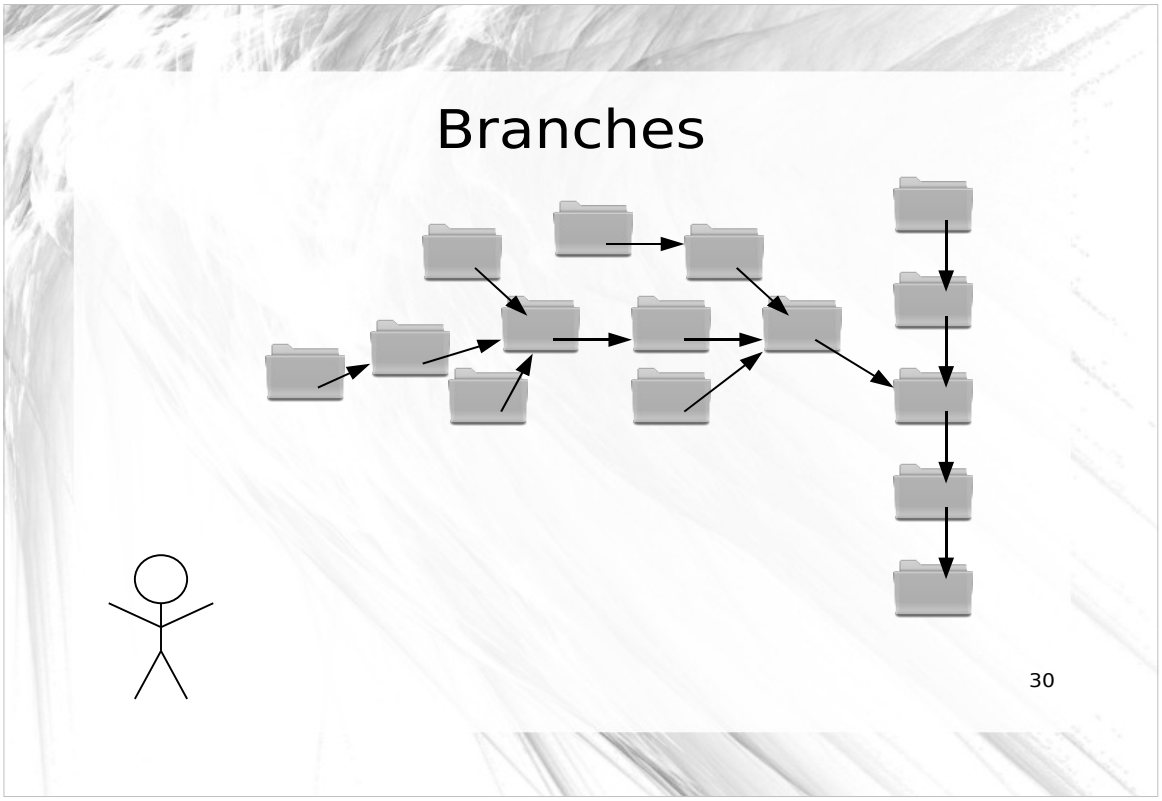
This organically produced structure allows for such great complexity, but the rules for finding your way back to the trunk are so simple, and perfect for keeping track of multiple lines of development!

It turns out that what they say about nature and creativity is true.



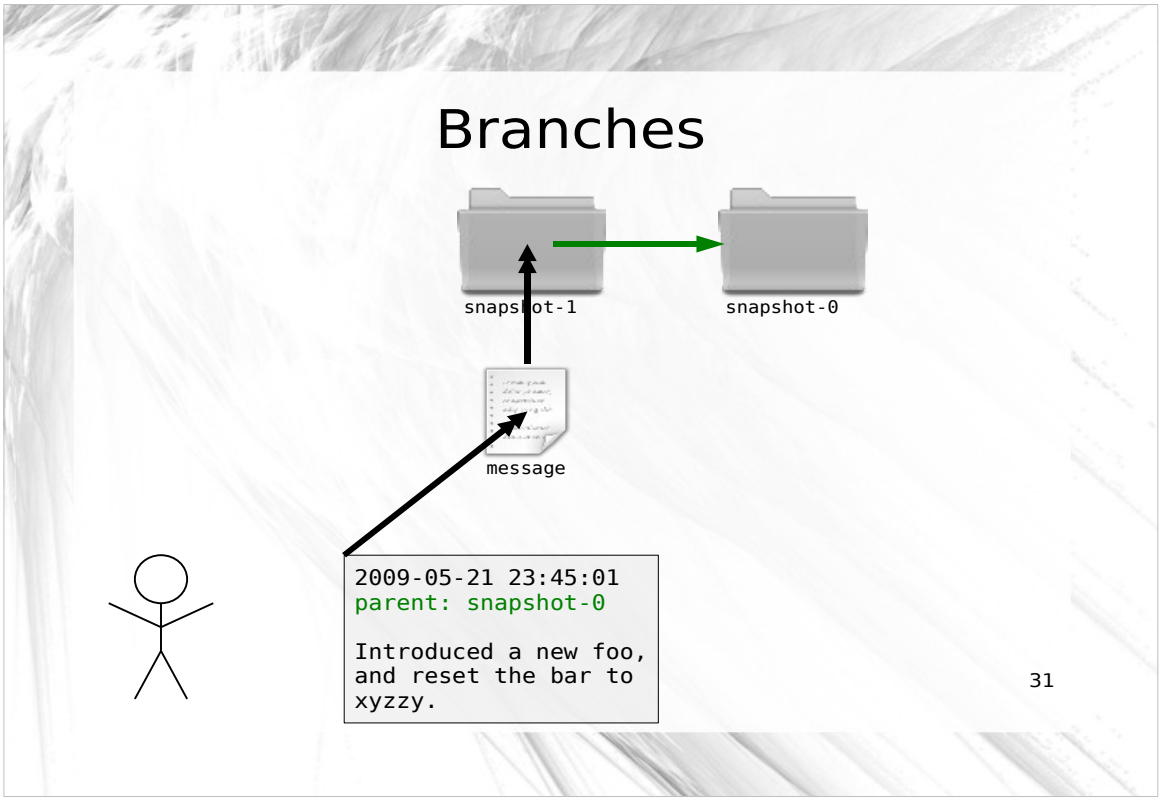

You have another epiphany.

By looking at your code history as a tree, solving the problem of ancestry becomes trivial.




You need each snapshot to point to the previous snapshot, also known as the “parent” snapshot.

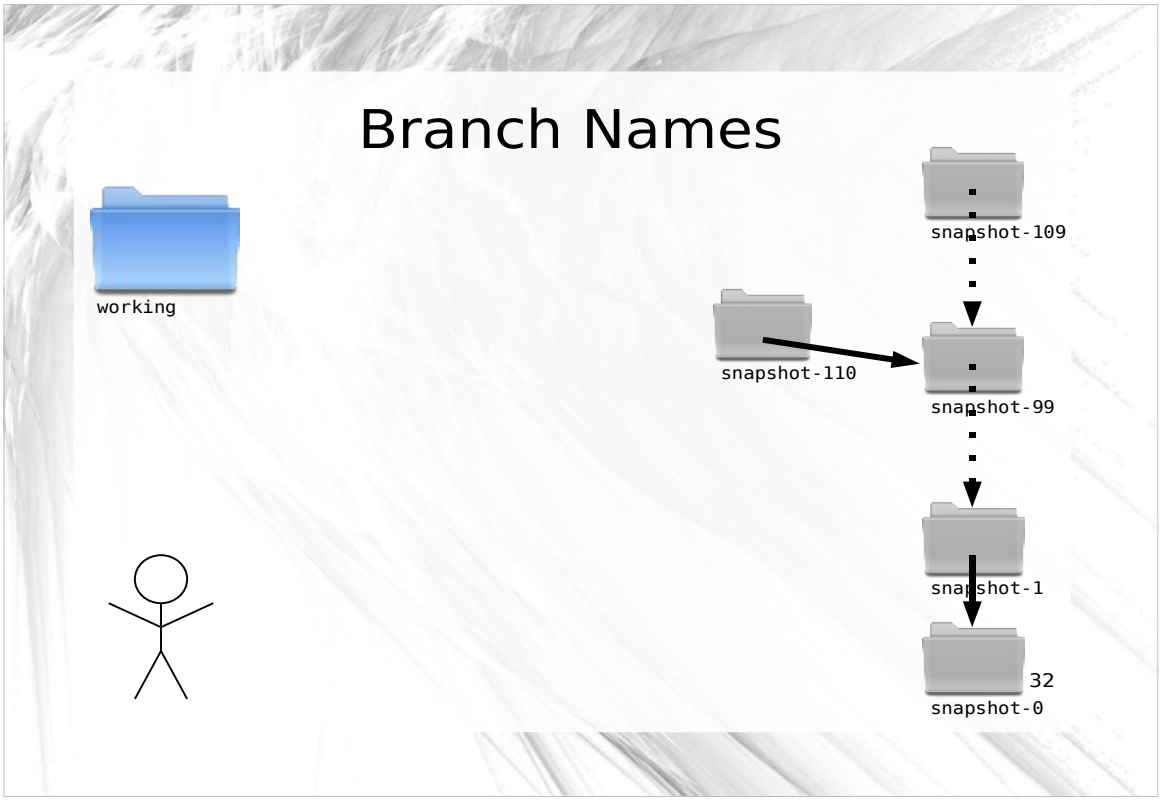
Each snapshot has one parent, except for the very first snapshot.



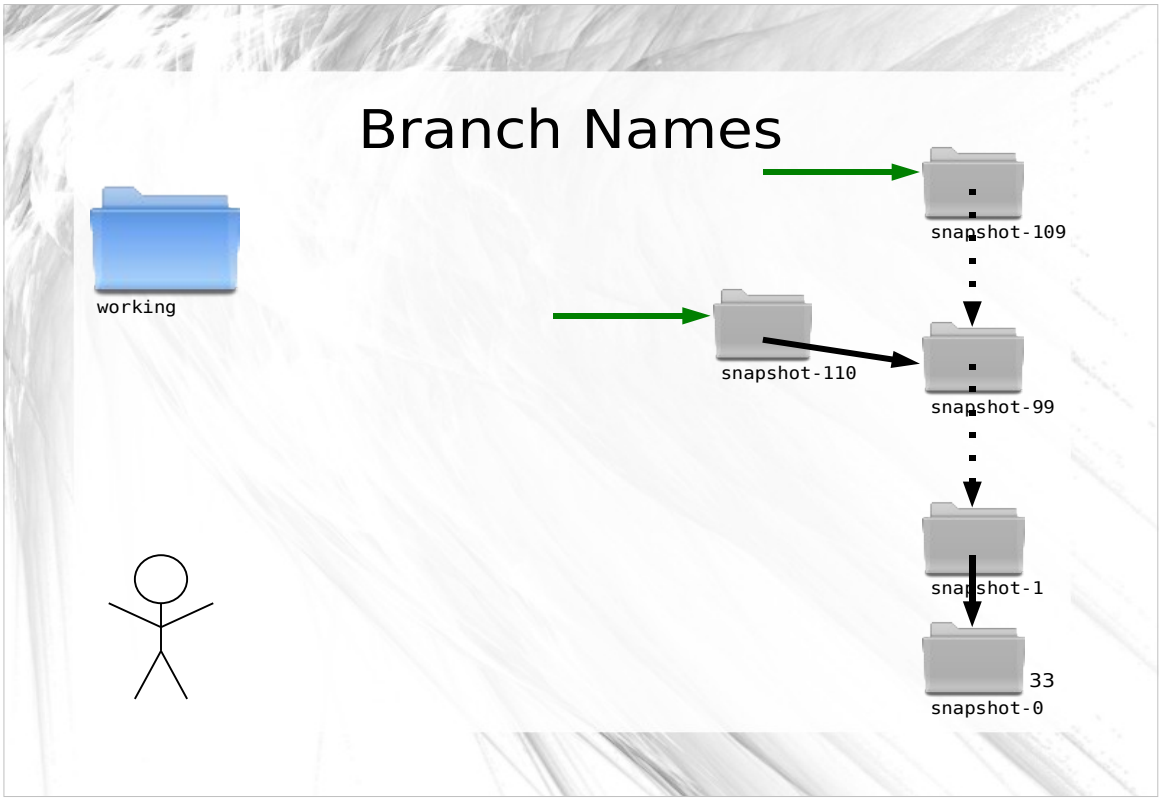

So how do you store this pointer?

All you need to do is include the name of the parent snapshot in the “message” file you write for each snapshot.

Adding that pointer enables you to easily and accurately trace the history of any given snapshot all the way back to the root.

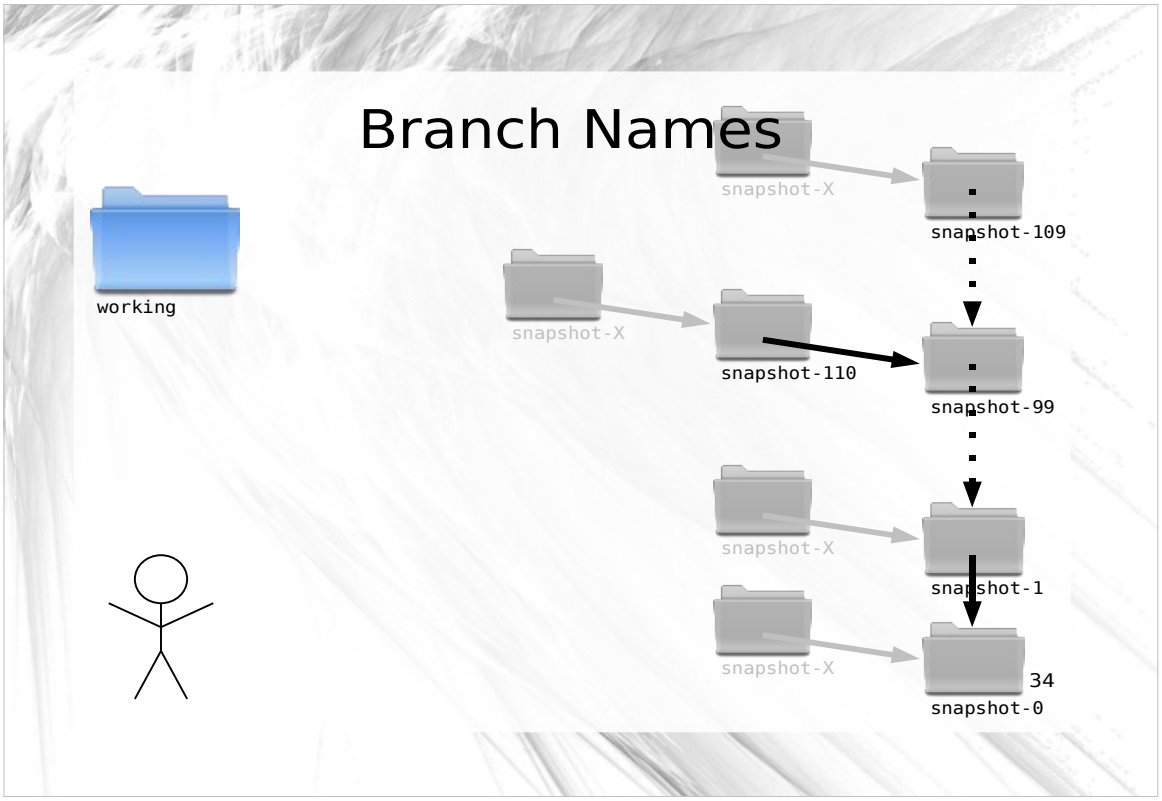



Your code history is now a tree.



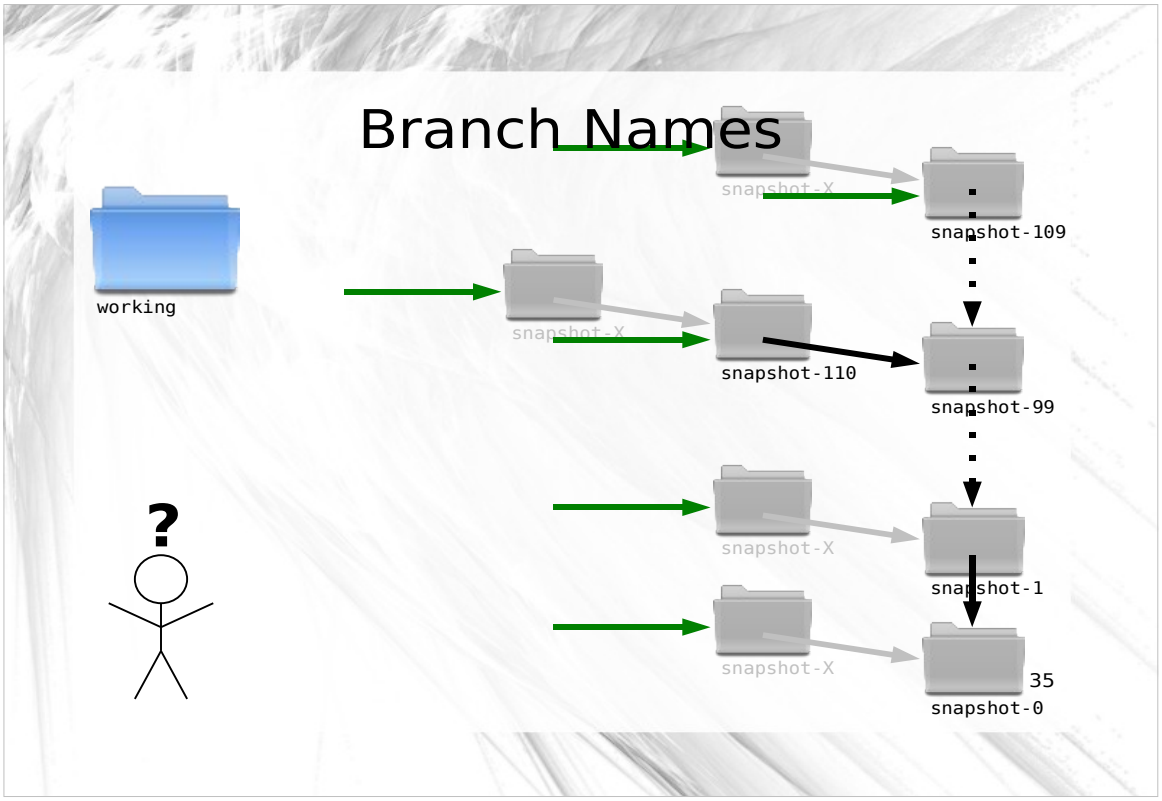

Instead of having a single latest snapshot, you have two: one for each branch.

Since you no longer have a linear system, your sequential numbering system has lost some of its usefulness: You cannot easily determine the latest snapshot, because you have multiple latest snapshots, one for each branch.




However, Creating new development branches has become so simple that you'll want to take advantage of it all the time.

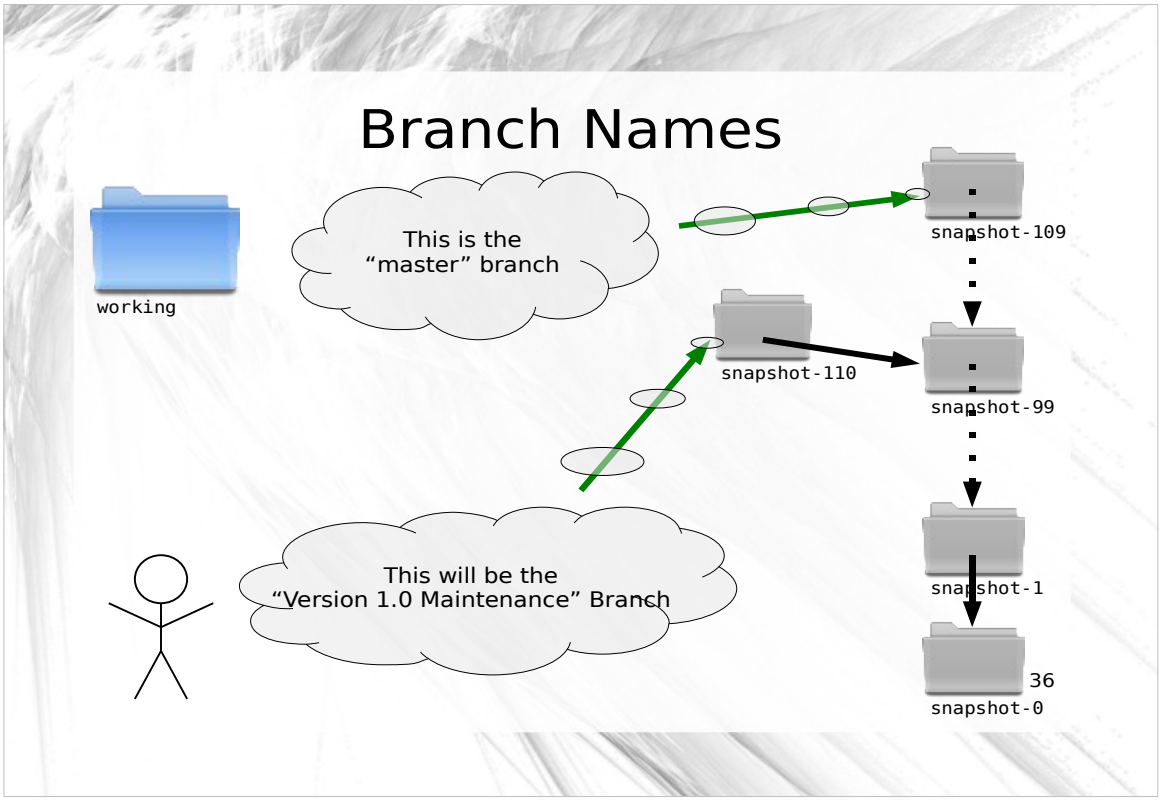
You'll be creating branches for fixes to old releases, for experiments that may not pan out; indeed it becomes possible to create a new branch for every feature you begin!




But like everything good in life, there is a price to be paid.

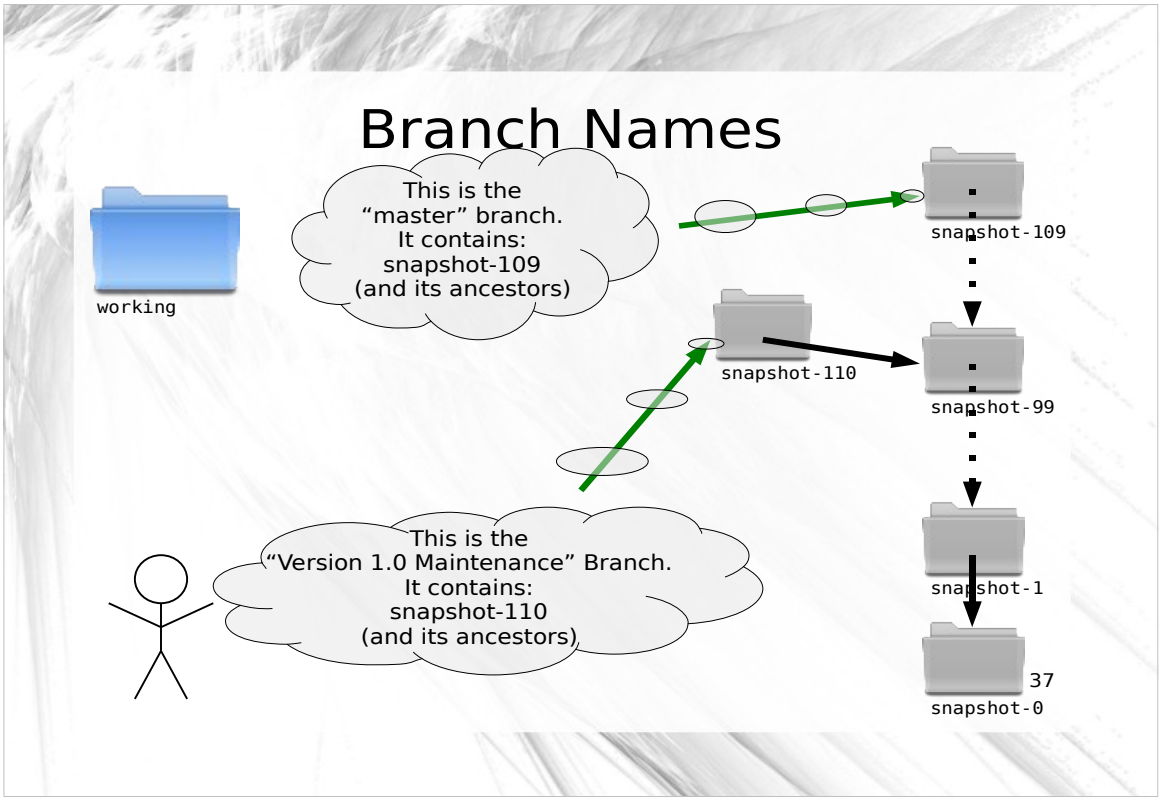
Each time you create a new snapshot, you must remember that the new snapshot becomes the latest on its branch.

Without this information, switching to a new branch would become a laborious process indeed.

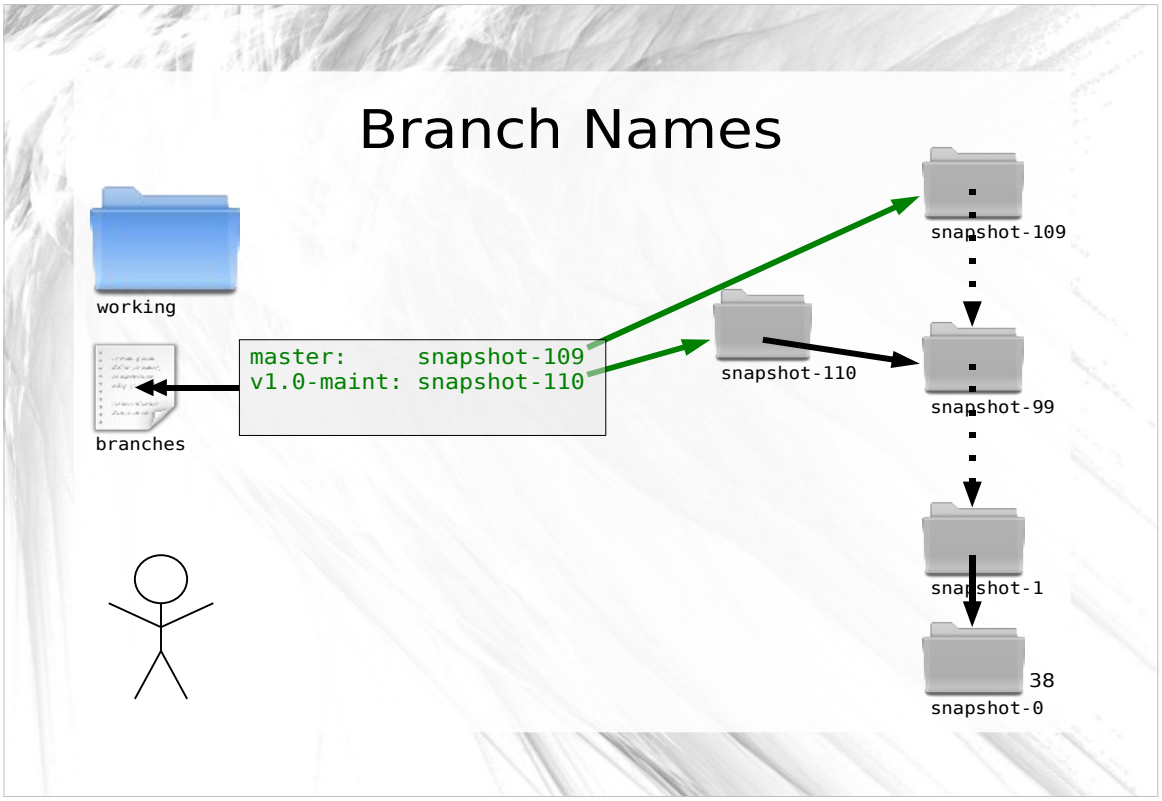



Every time you create a new branch you probably give it a name in your head.

“This will be the Version 1.0 Maintenance Branch,” you might say. Perhaps you refer to the former linear branch as the “master” branch.



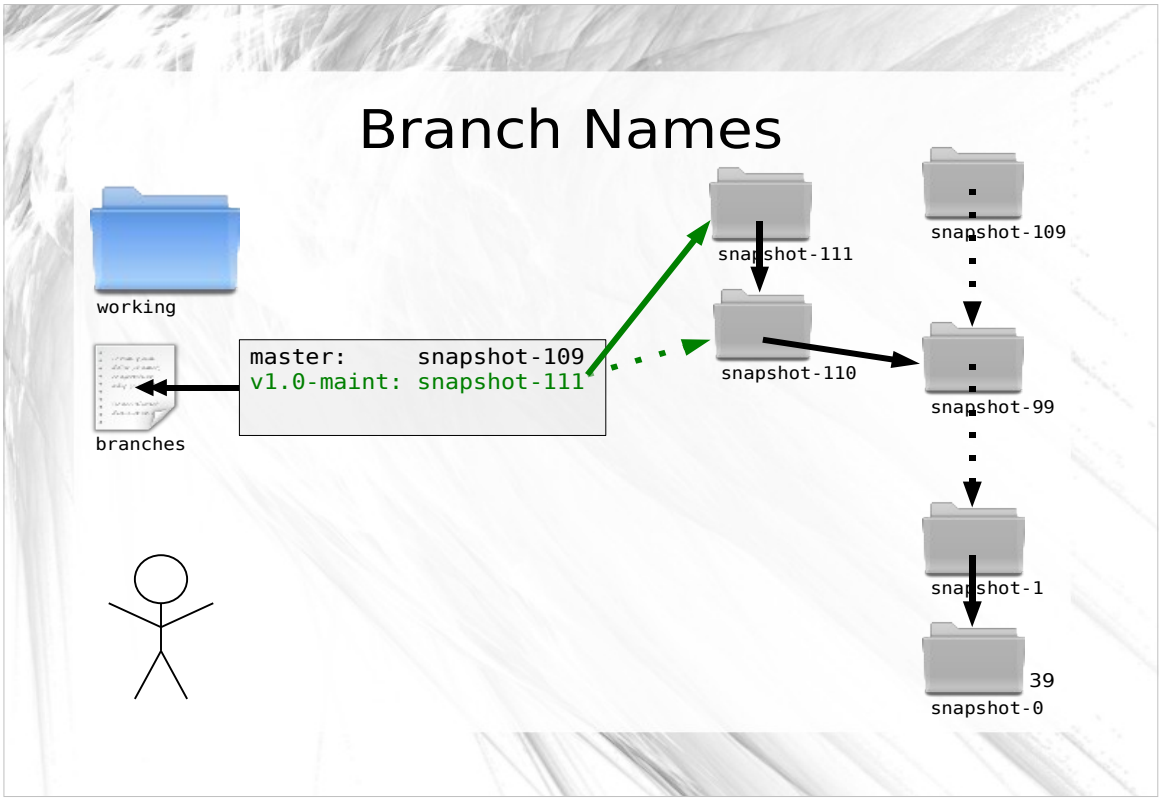

Now, to find to the snapshots on a branch, you only need to remember the latest snapshot on that branch. The earlier snapshots is found by following the parent pointer from each snapshot.




Storing the branch names is trivial.

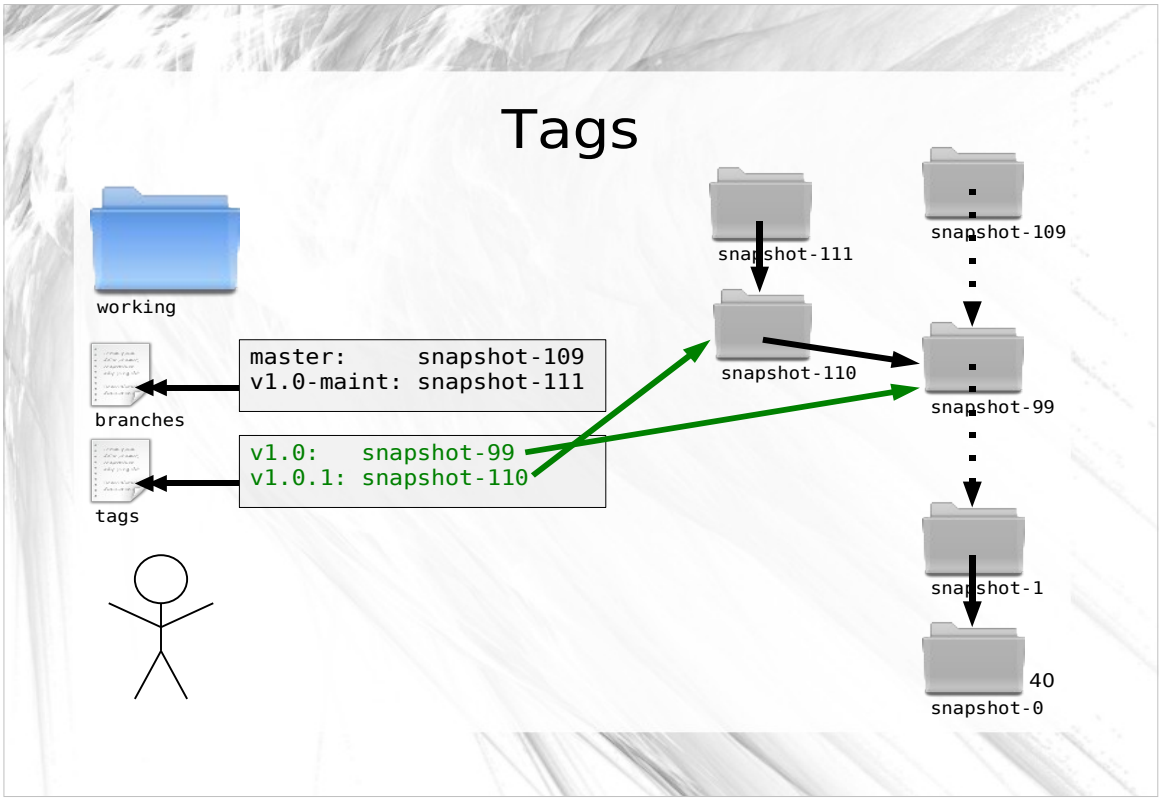
In a file named “branches”, stored outside of any specific snapshot, you simply list the name/snapshot pairs that represent the tips of branches.

To switch to a named branch you need only look up the snapshot for the corresponding name from this file.




Because you're only storing the latest snapshot on each branch, creating a new snapshot now contains an additional step:

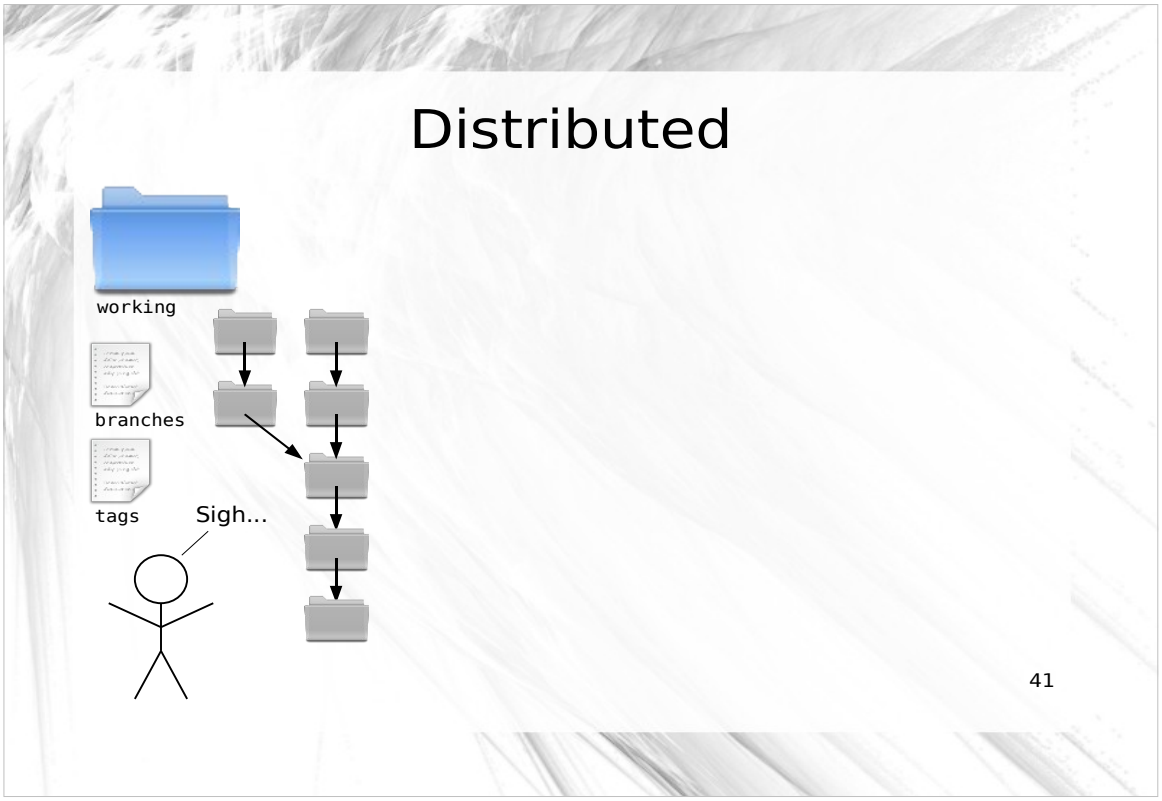
If the new snapshot is being created as part of a branch, the "branches" file must be updated so that the name of the branch becomes associated with the new snapshot. A small price to pay for the benefit.




After using branches for a while you notice that you also need a different kind of branch. You want a branch that always points to a certain snapshot, and never moves.

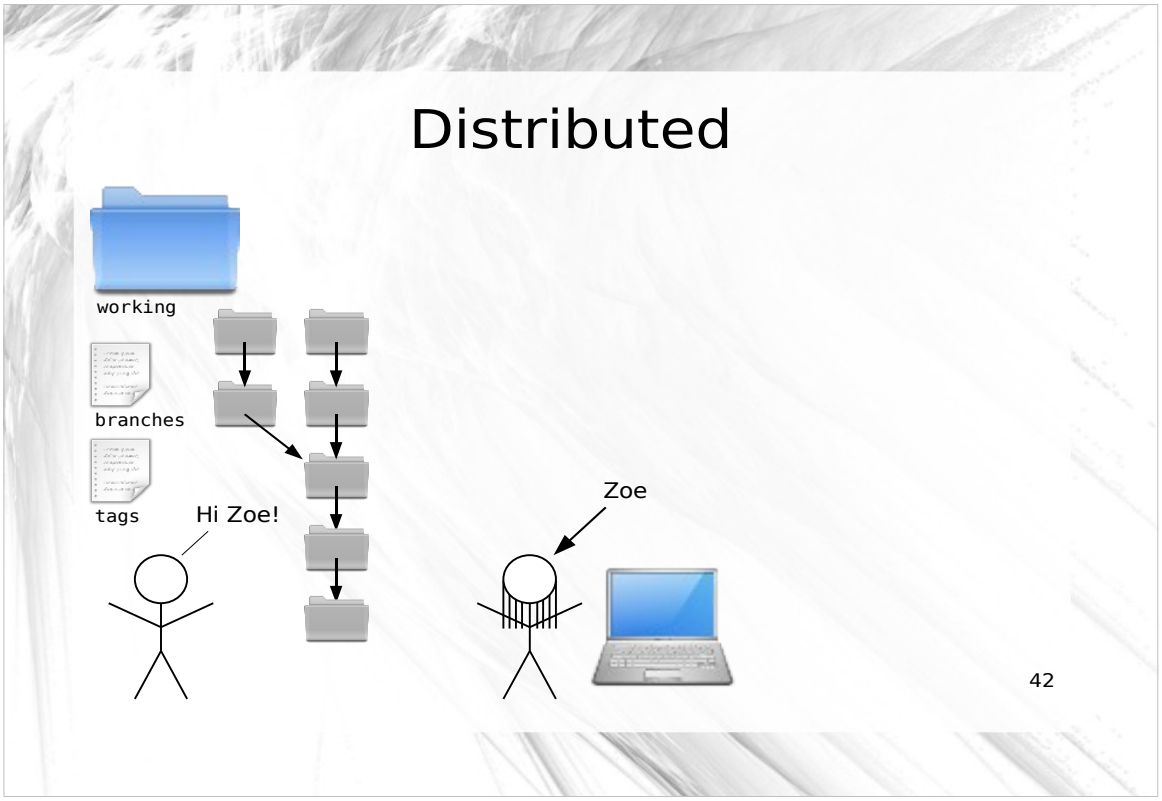
You want to use these special branches for labeling certain snapshots of interest, like “Version 1.0” and “Version 1.0.1”.

Since they don't behave like regular branches, you decide to call them “tags”, and store them in a separate file, so that you don't accidentally treat them as a regular branch.

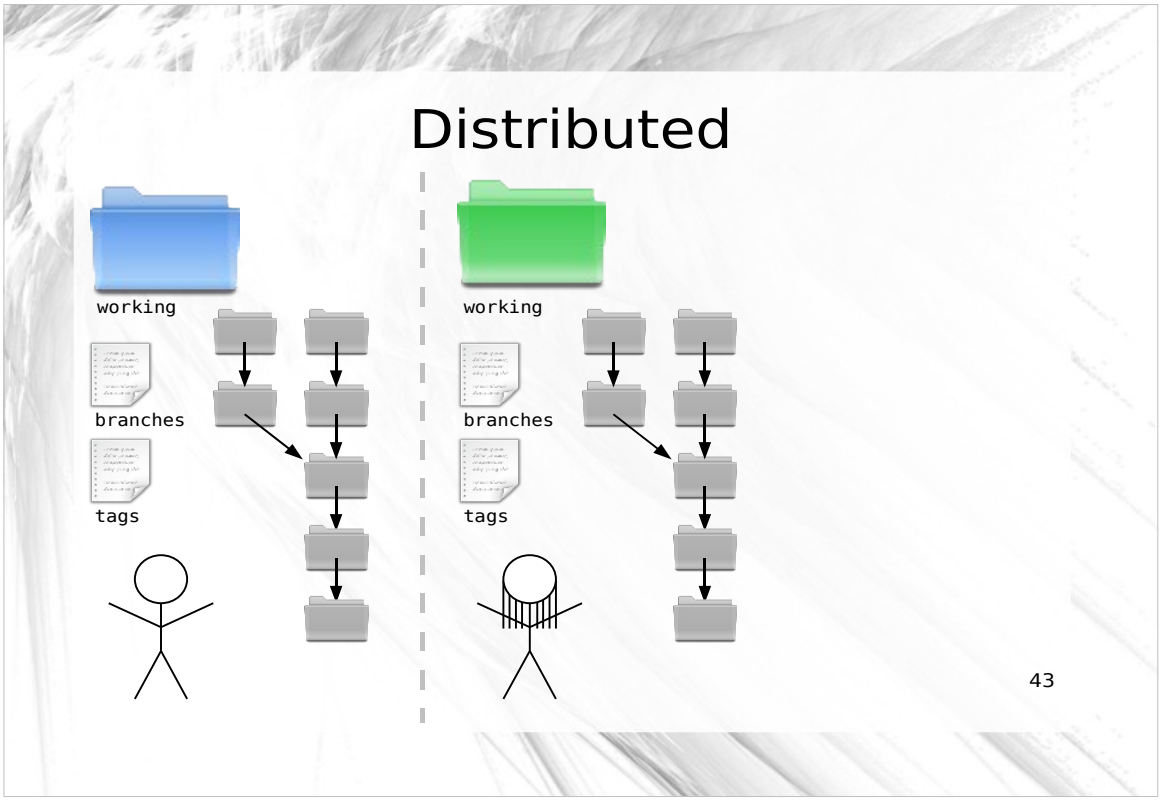



Working on your own gets pretty lonely.

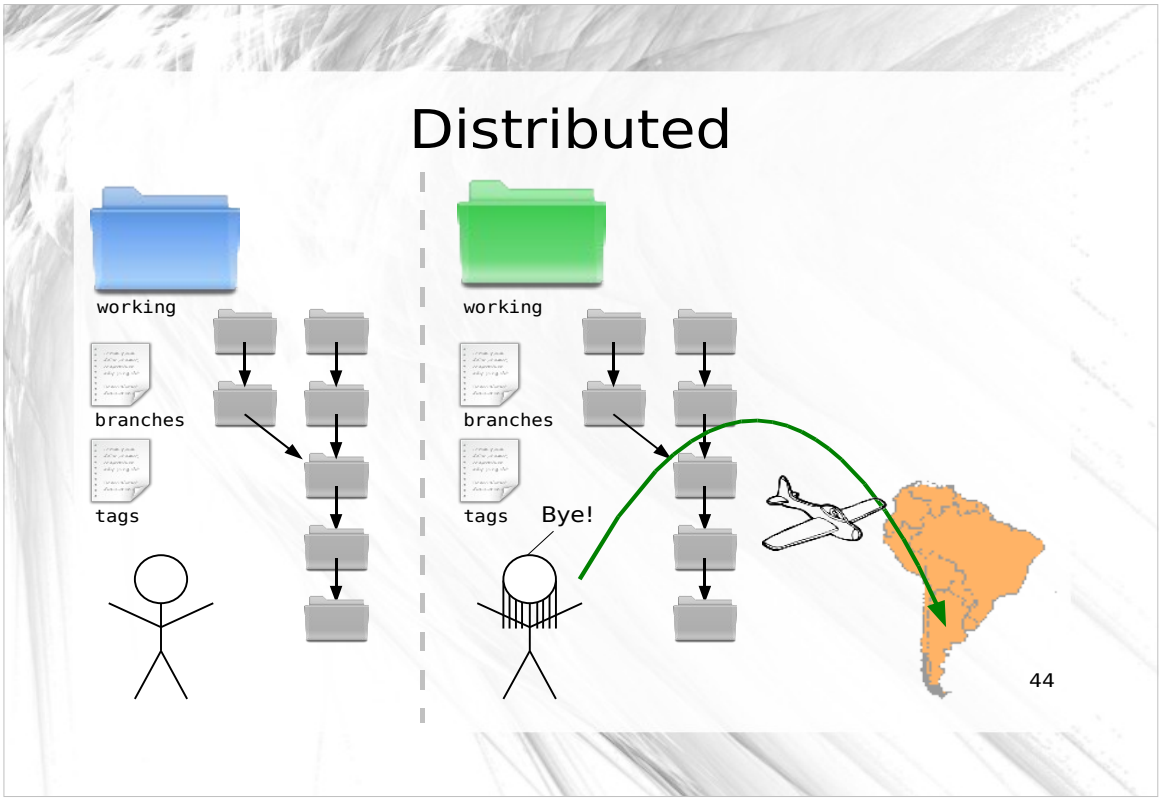
Wouldn’t it be nice if you could invite a friend to work on your project with you?




Well, you’re in luck. Your friend Zoe has a computer setup just like yours and wants to help with the project.

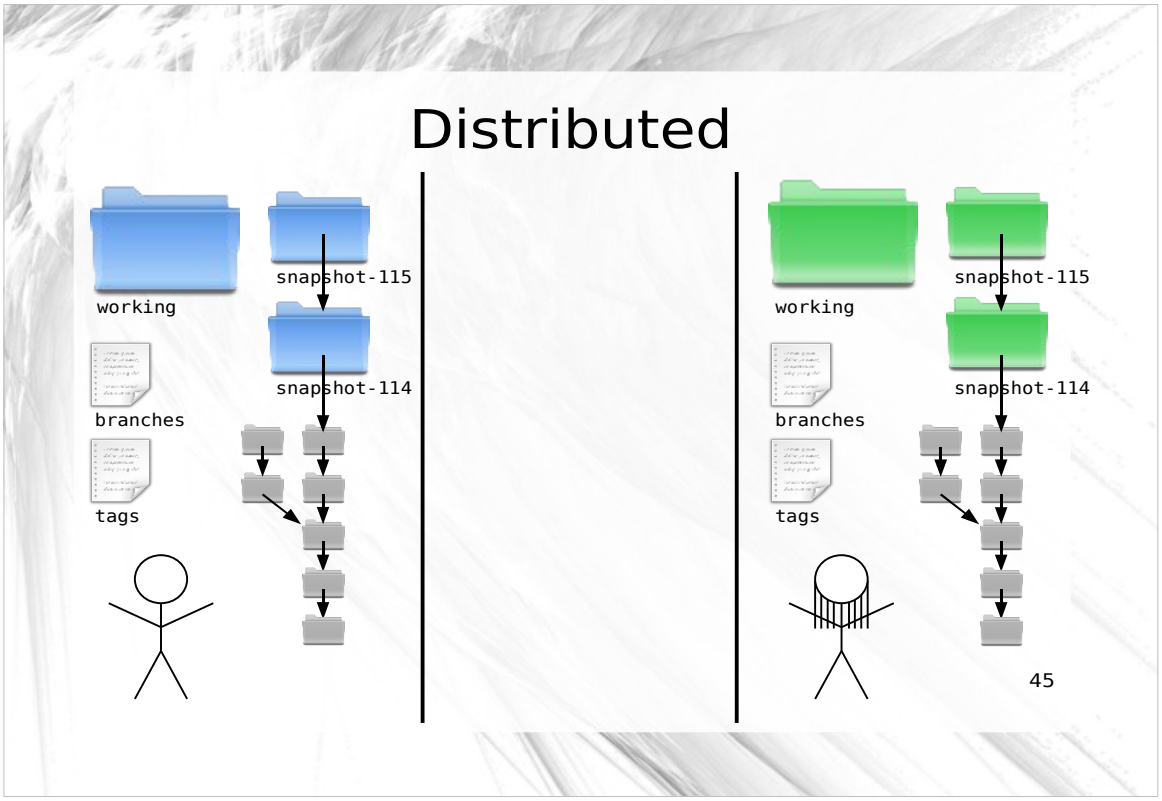



Because you've created such a great version control system, you tell her all about it and send her a copy of all your snapshots, branches, and tags so she can enjoy the same benefits of the code history.




It’s great to have Zoe on the team but she has a habit of taking long trips to far away places without internet access.

As soon as she has the source code, she catches a flight to Patagonia and you don’t hear from her for a week.

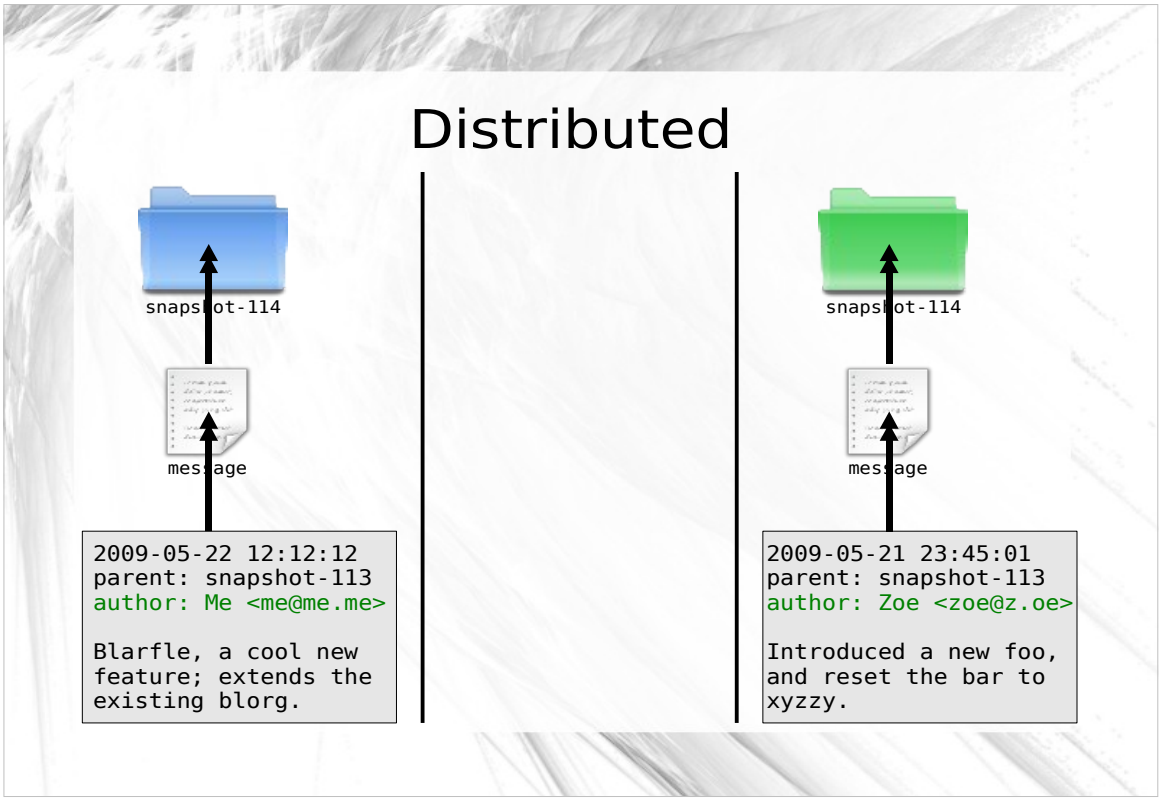



In the meantime you both code up a storm. When she finally gets back, you discover a critical flaw in your VCS.

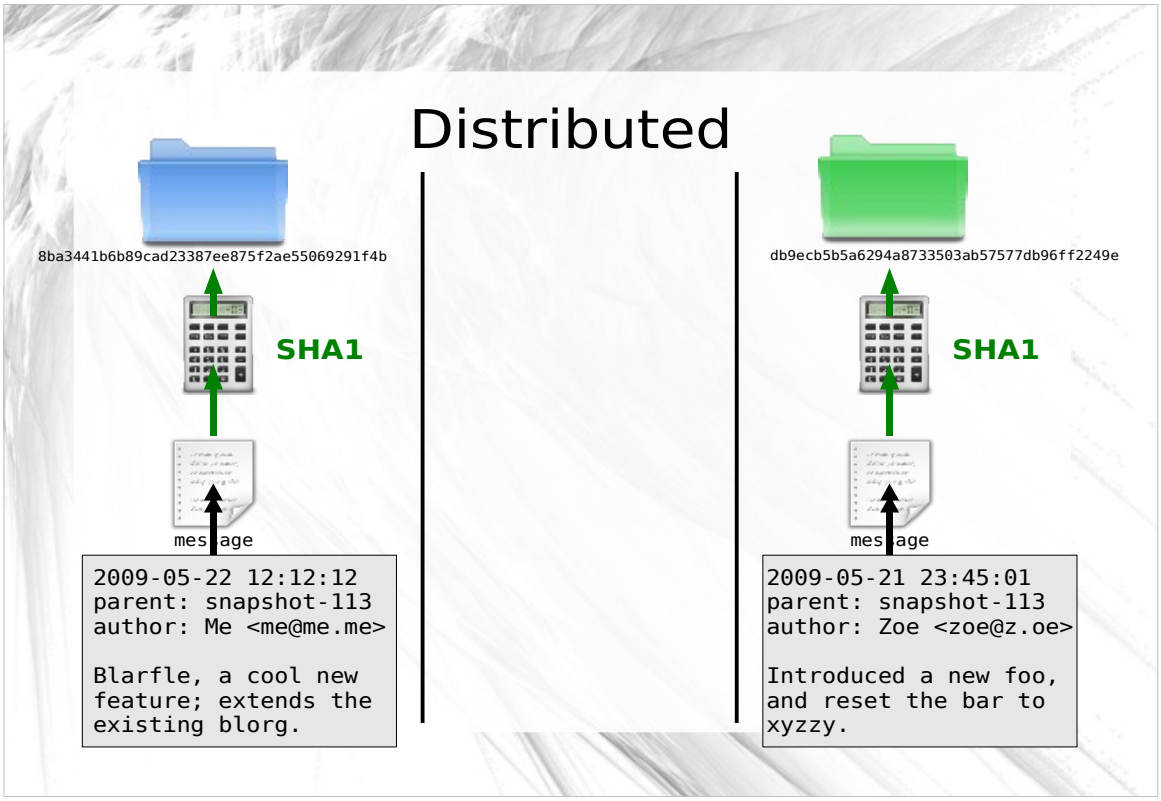
Because you've both been using the same numbering system, you each have directories named "snapshot-114", "snapshot-115", and so on, but with different contents!

To make matters worse, you don't even know who authored the changes in those new snapshots.

Together, you devise a plan for dealing with these problems.




First, snapshot messages will henceforth contain author name and email.

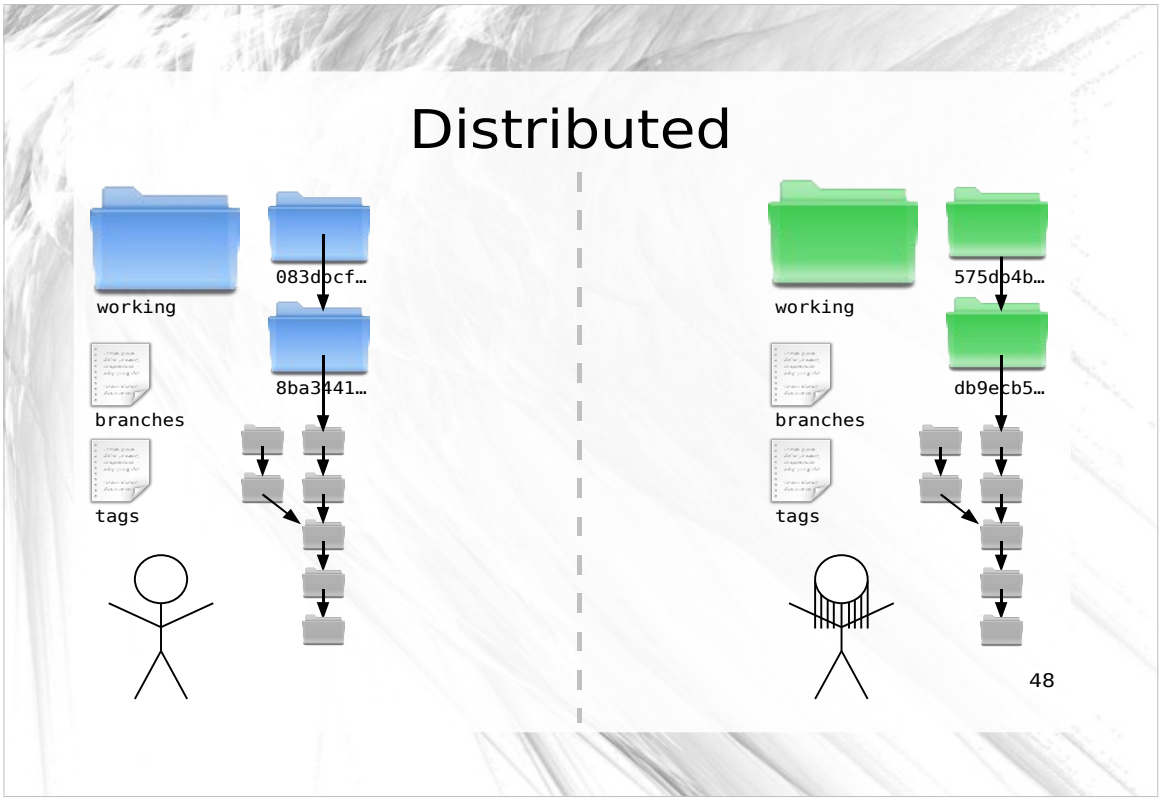



Second, snapshots will no longer be named with simple numbers. Instead, you'll use the contents of the "message" file to produce a hash.

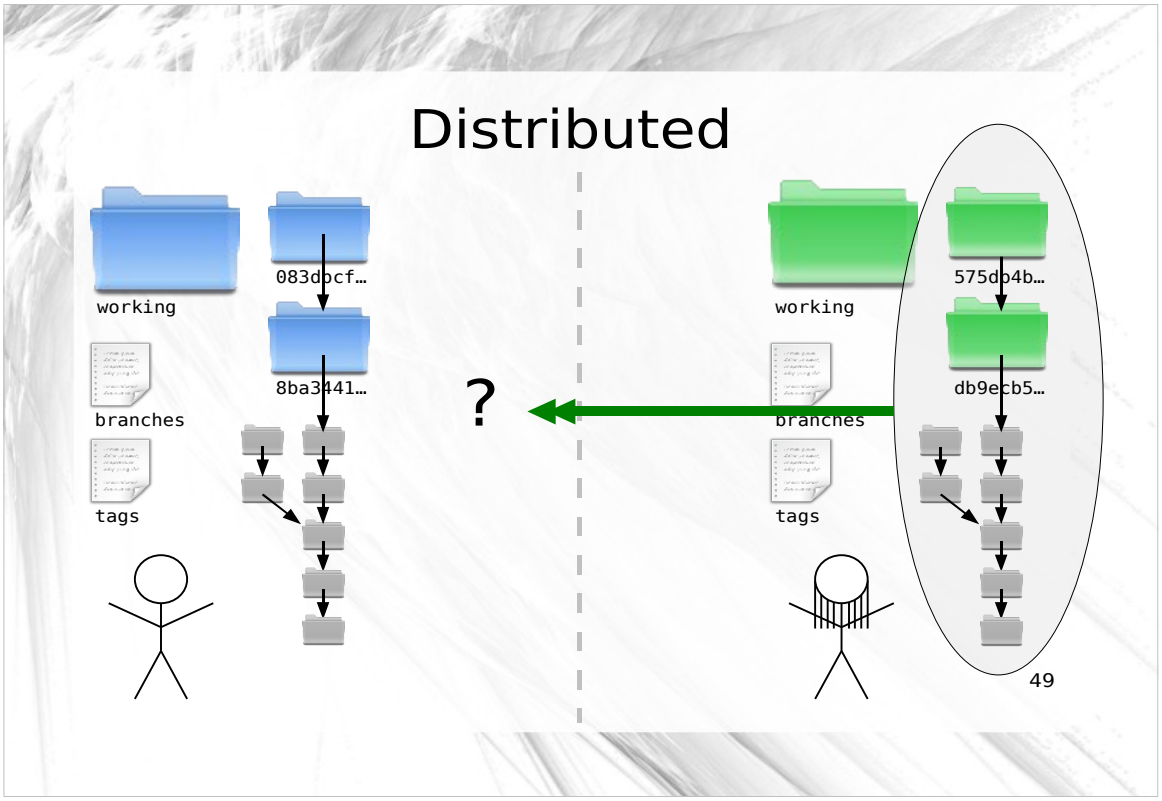
This hash will be guaranteed to be unique to the snapshot since no two messages will ever have the same date, message, parent, and author.

To make sure everything goes smoothly, you both agree to use the SHA1 hash algorithm that takes the contents of a file and produces a 40 character hexadecimal string.

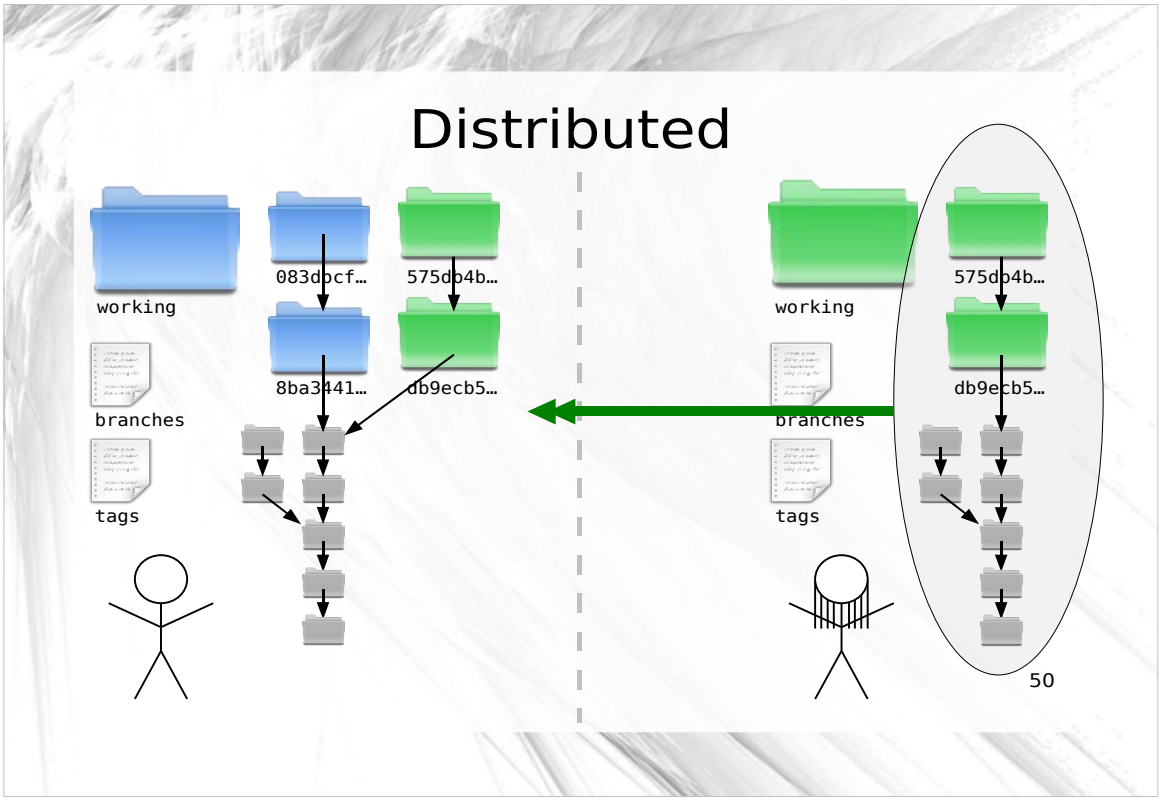
You both update your histories with the new technique and instead of clashing "snapshot-114" folders, you now have distinct folders named "8ba3441b6b89cad23387ee875f2ae55069291f4b" and "db9ecb5b5a6294a8733503ab57577db96ff2249e".




Of course, you update all the other snapshots as well...

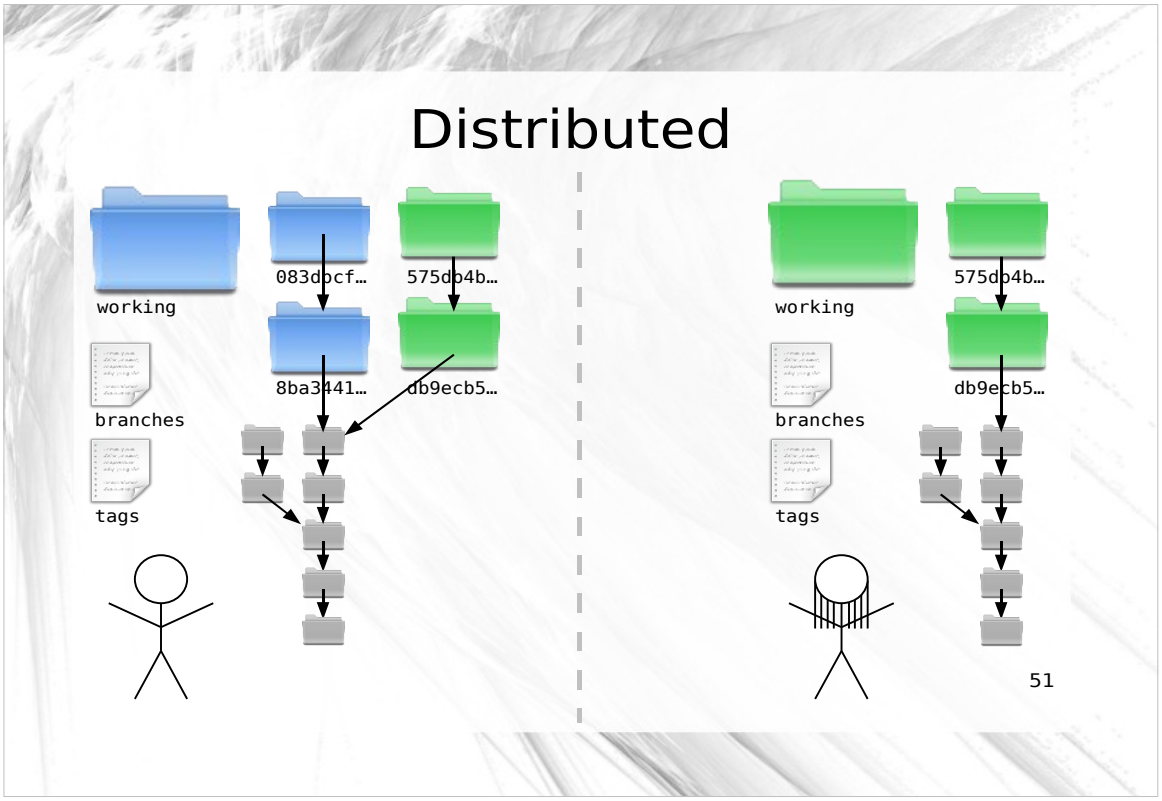



With the updated naming scheme, it becomes trivial for you to fetch all the new snapshots from Zoe’s computer and place them next to your existing snapshots.




Because every snapshot specifies its parent, and identical messages (and therefore identical snapshots) have identical names no matter where they are created, the history of the codebase can still be drawn as a tree.

Only now, the tree is comprised of snapshots authored by both Zoe and you.

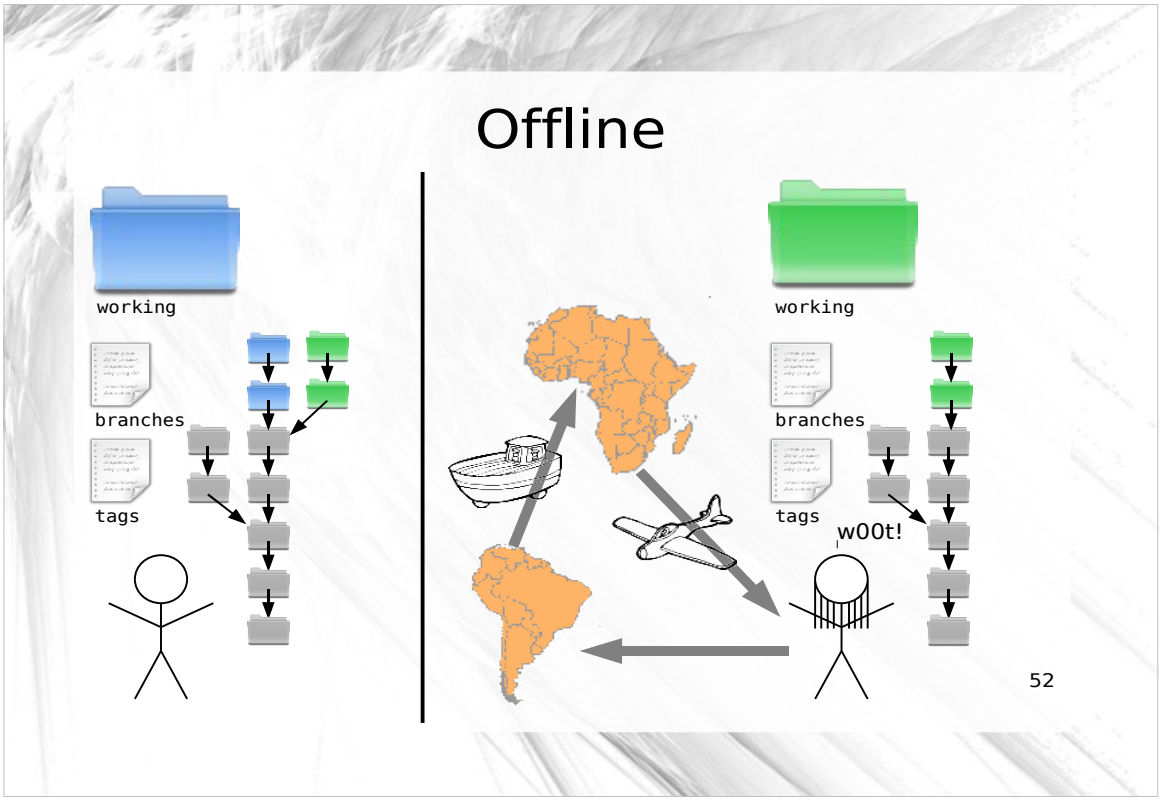



This point is important, so I'll repeat it:

A snapshot is identified by a SHA1 that uniquely identifies it (and its parent).

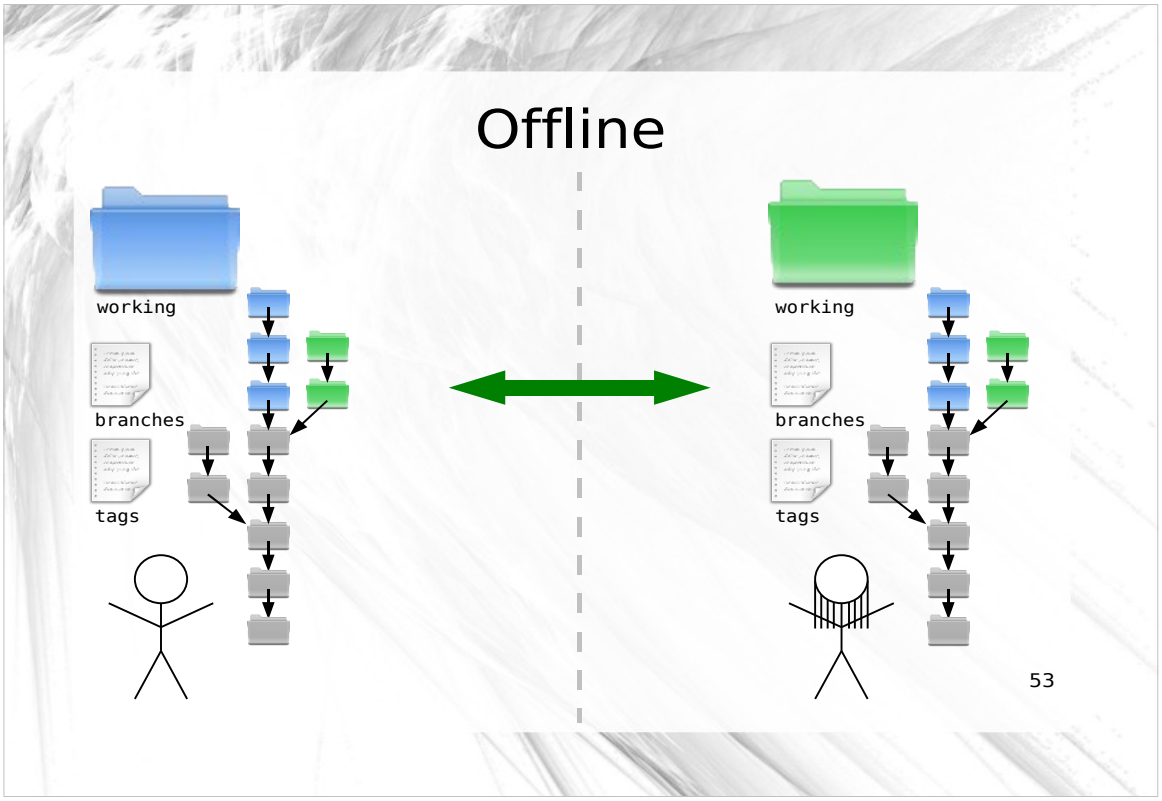
These snapshots can be created and moved around between computers without losing their identity or where they belong in the history tree of a project.

What’s more, snapshots can be shared or kept private as you see fit. If you have some experimental snapshots that you want to keep to yourself, you can do so quite easily. Just don’t make them available to Zoe!

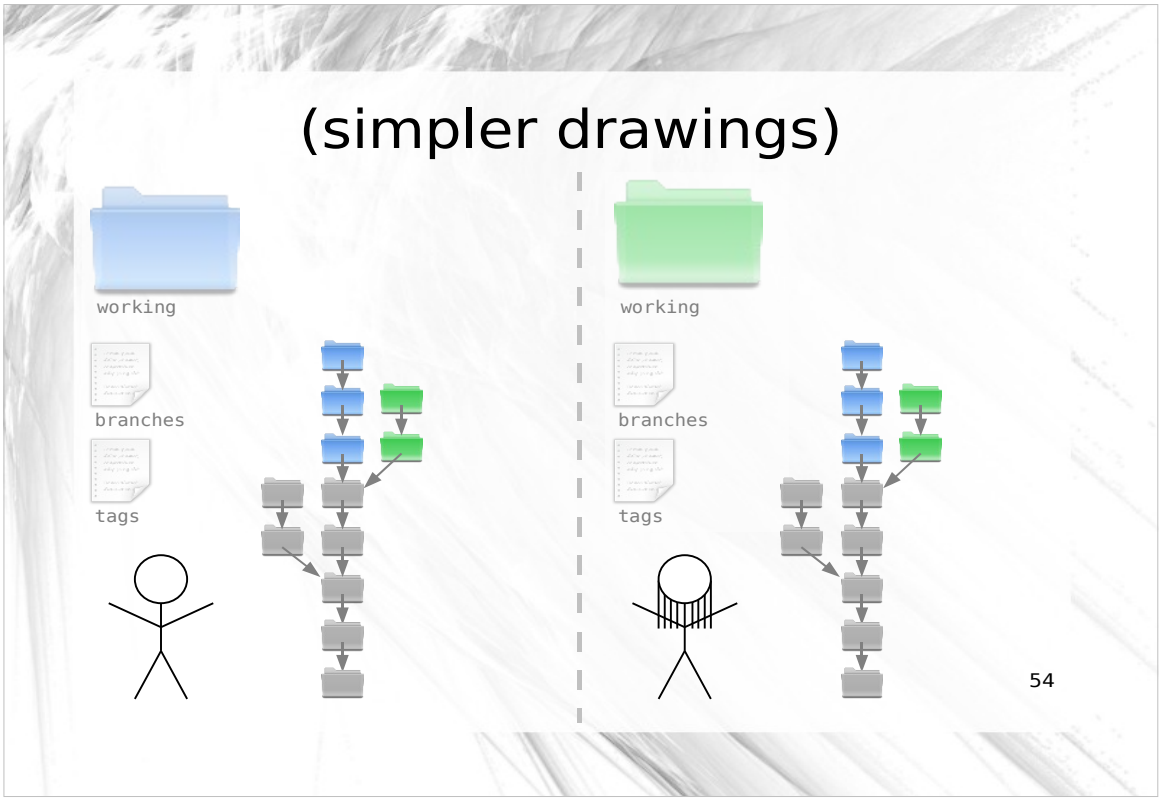



Zoe’s travel habits cause her to spend countless hours on airplanes and boats. Most of the places she visits have no readily available internet access. At the end of the day, she spends more time offline than online.

It’s no surprise, then, that Zoe raves about your VCS. All of the day to day operations that she needs to do can be done locally.

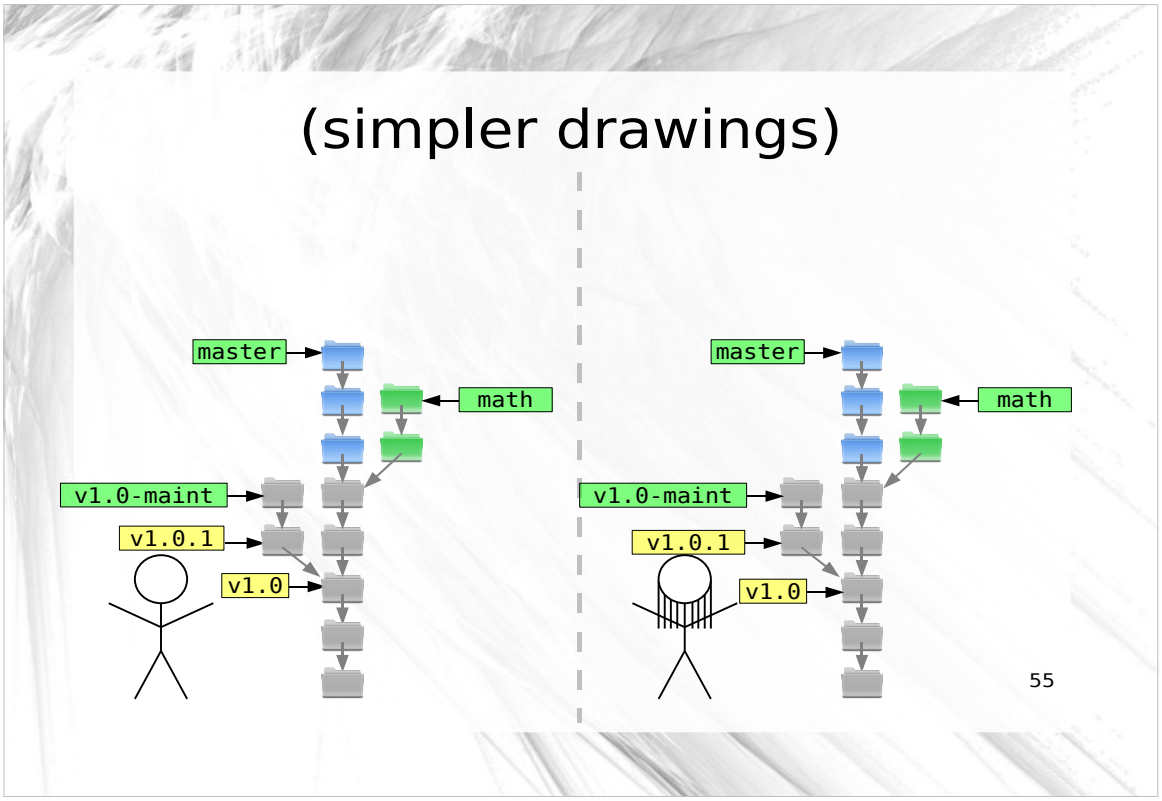



The only time she needs a network connection is when she’s ready to share her snapshots with you.




Now, let's simplify our drawings a little, so that we can focus on the important issues:

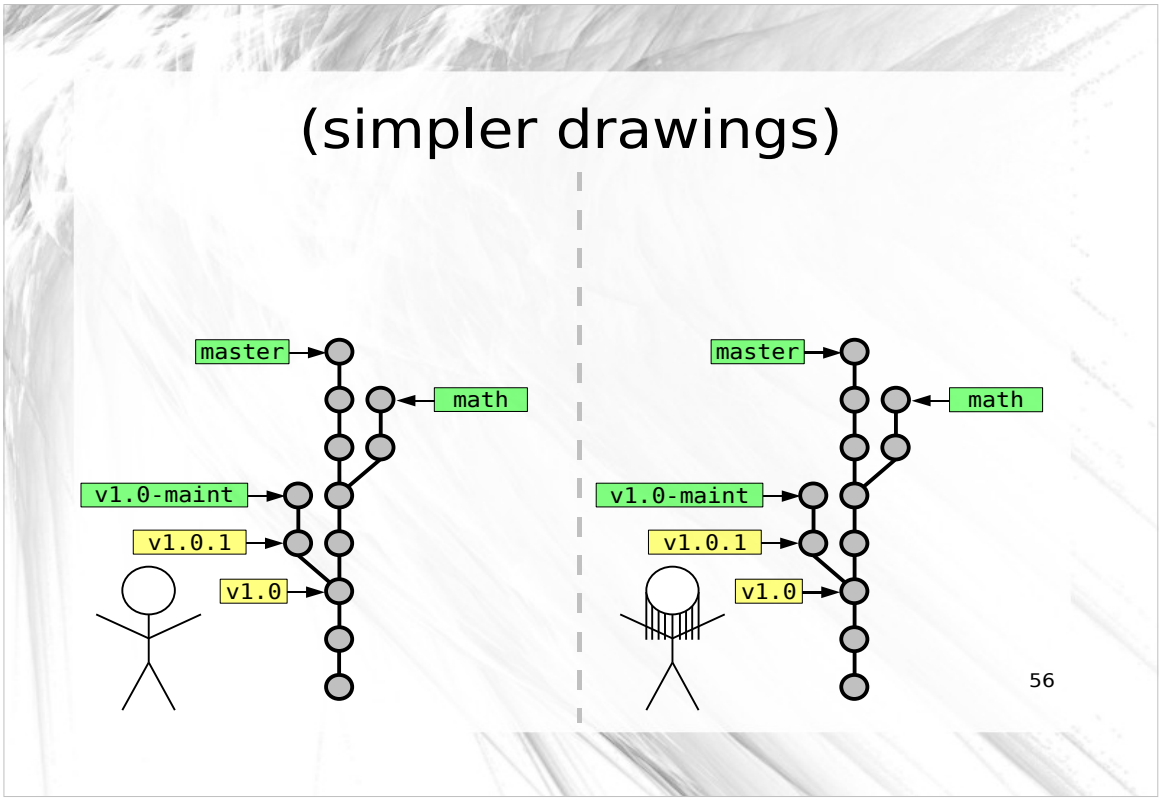
We don't need to draw the “working” directory, and the “branches” and “tags” files. We know they're always there, anyway.




Instead we will add labels that that point to the branches and tags that we are currently interested in.

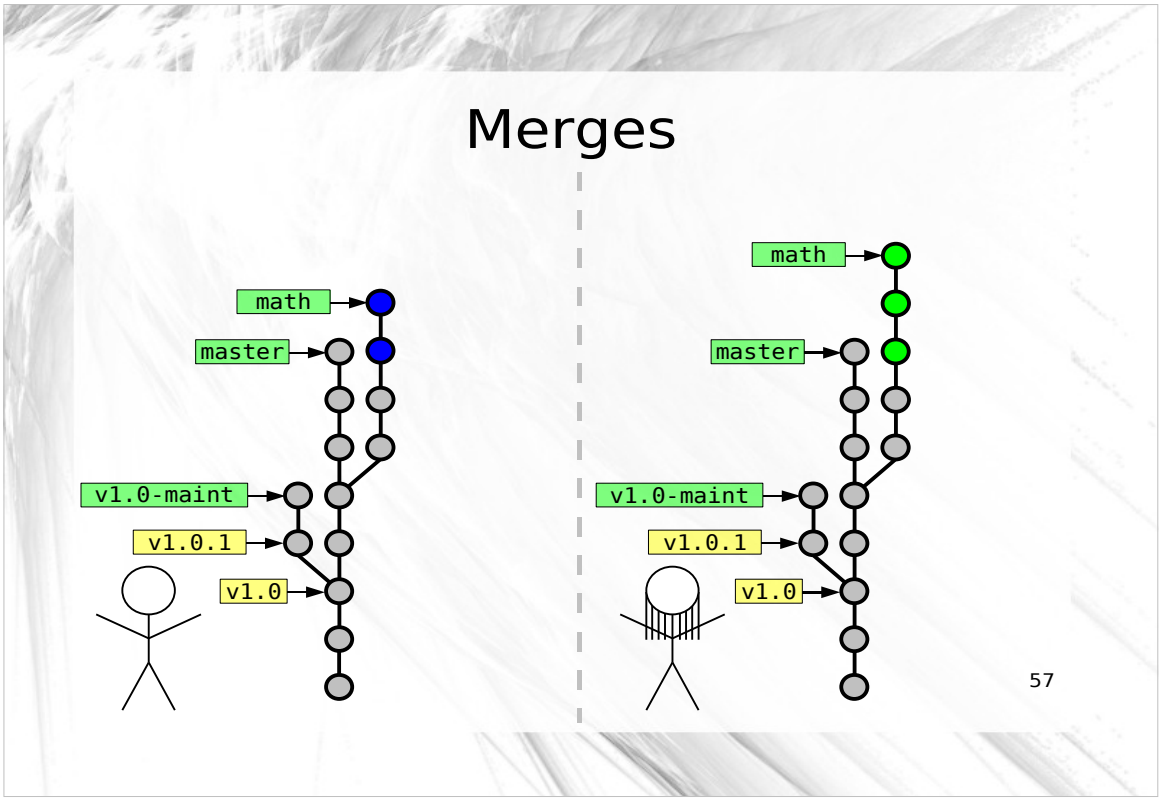
Branches are green, and tags are yellow.

There's a “master” branch, and a “v1.0-maint” branch, and two tags - “v1.0” and “v1.0.1”.



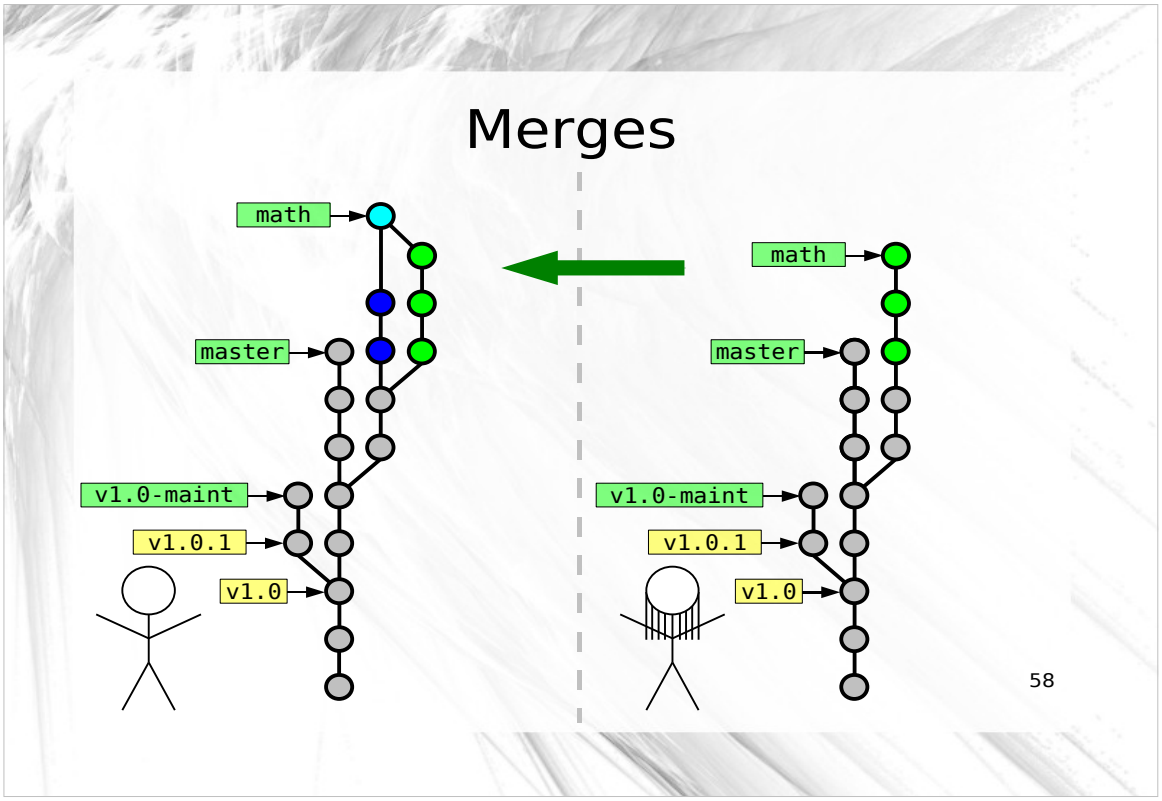

Finally, let's draw the snapshots as simple dots.

And since the parent pointers always point downwards in this graph, we can use simple lines instead.




Before Zoe left on her trip, you had asked her to start working off of the branch named ‘math’ and to implement a function that generated prime numbers.

Meanwhile, you were also developing off of the ‘math’ branch, only you were writing a function to generate magic numbers.

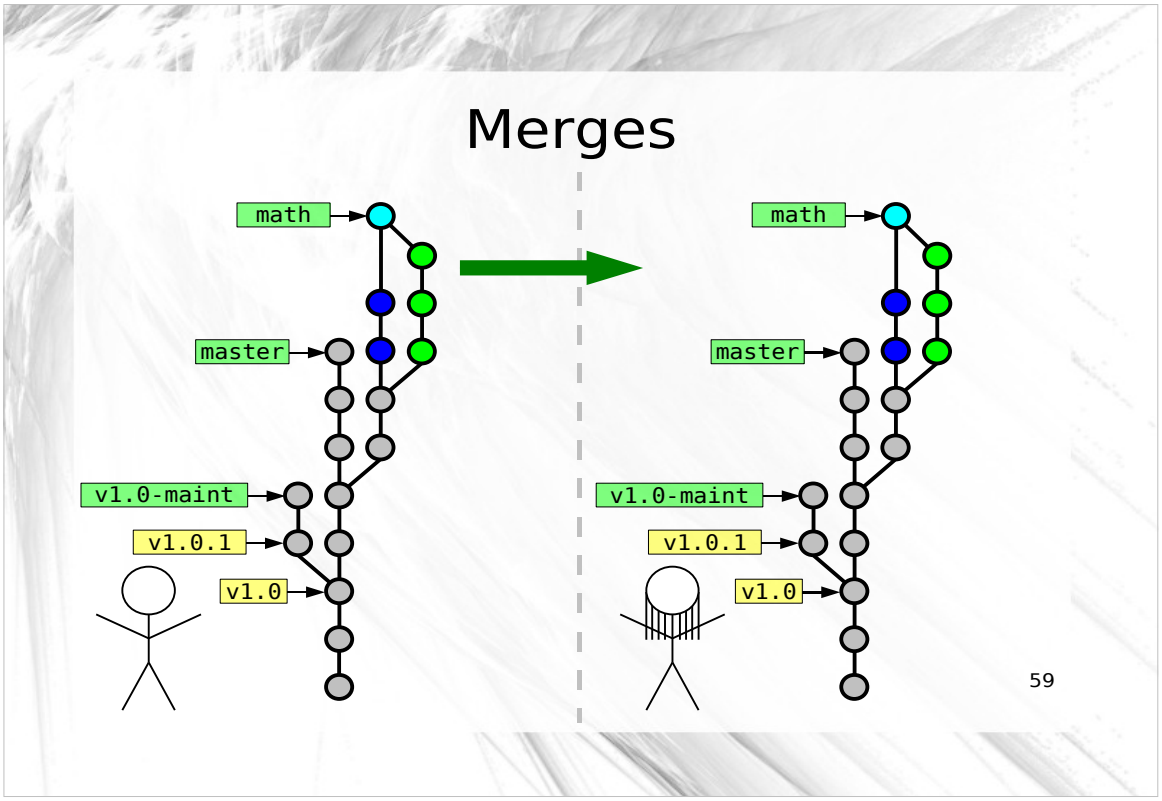



Now that Zoe has returned, you are faced with the task of merging these two separate branches of development into a single snapshot.

The merge snapshot doesn’t contain any others changes except reconciling the two branches into a single snapshot. Since you both worked on separate tasks, the merge is simple.

While constructing the snapshot message for the merge, you realize that this snapshot is special. Instead of just a single parent, this merge snapshot has two parents!

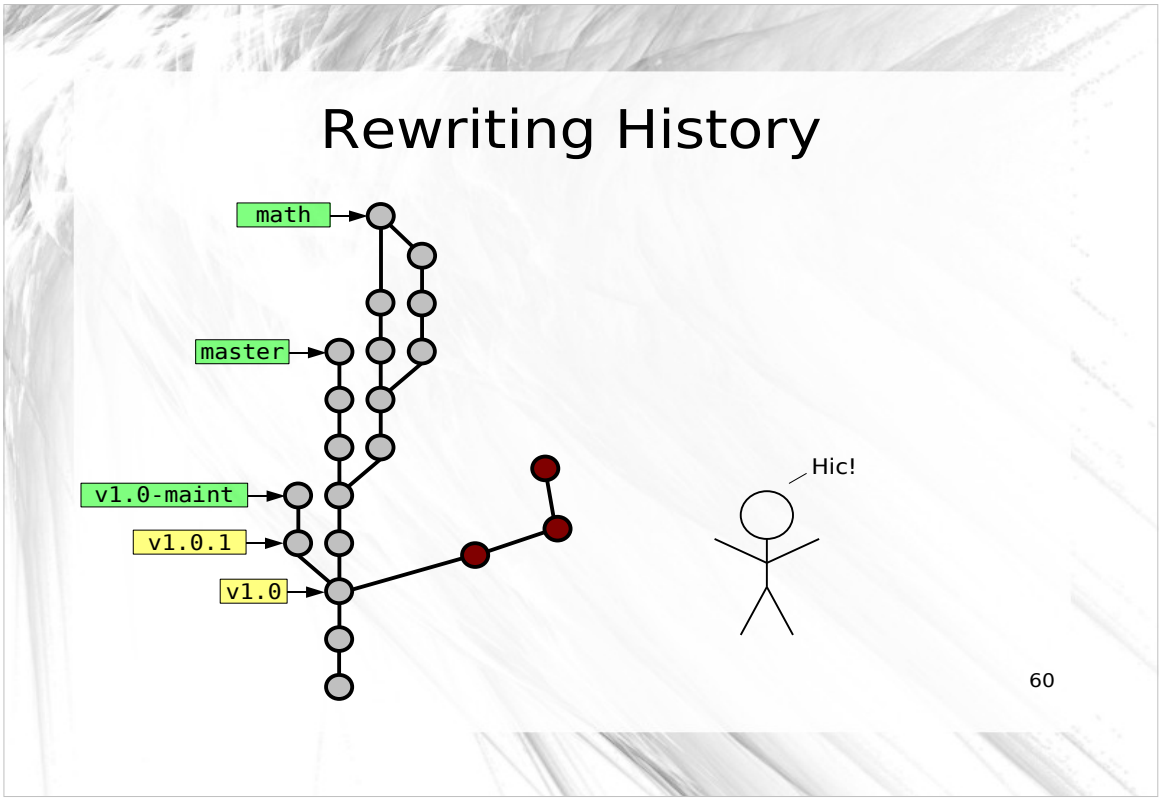
The first parent is the latest on your ‘math’ branch and the second parent is Zoe’s latest on her ‘math’ branch.




Once you complete the merge, Zoe fetches all the snapshots that you have that she does not:

- Your development on the ‘math’ branch
- Your merge snapshot.

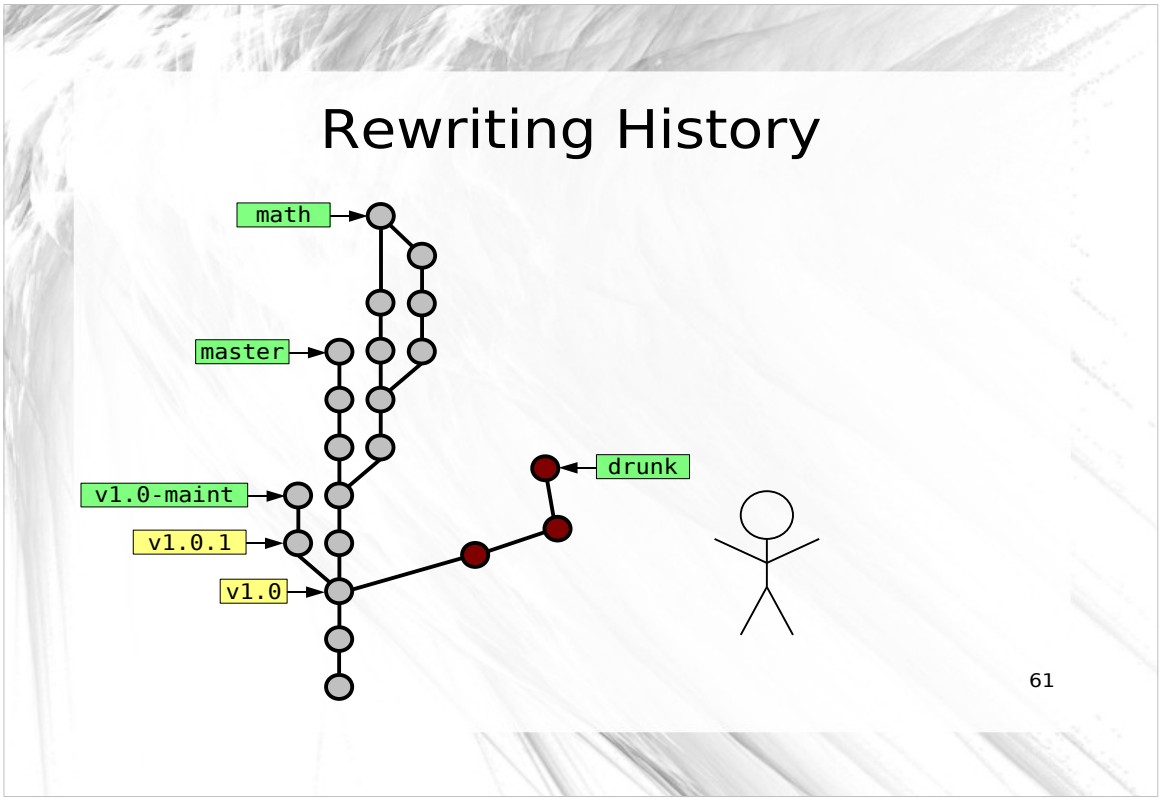
Once she does this, both of your histories match exactly!




Like many software developers you have a compulsion to keep your code clean and very well organized. This carries over into a desire to keep your code history well groomed.

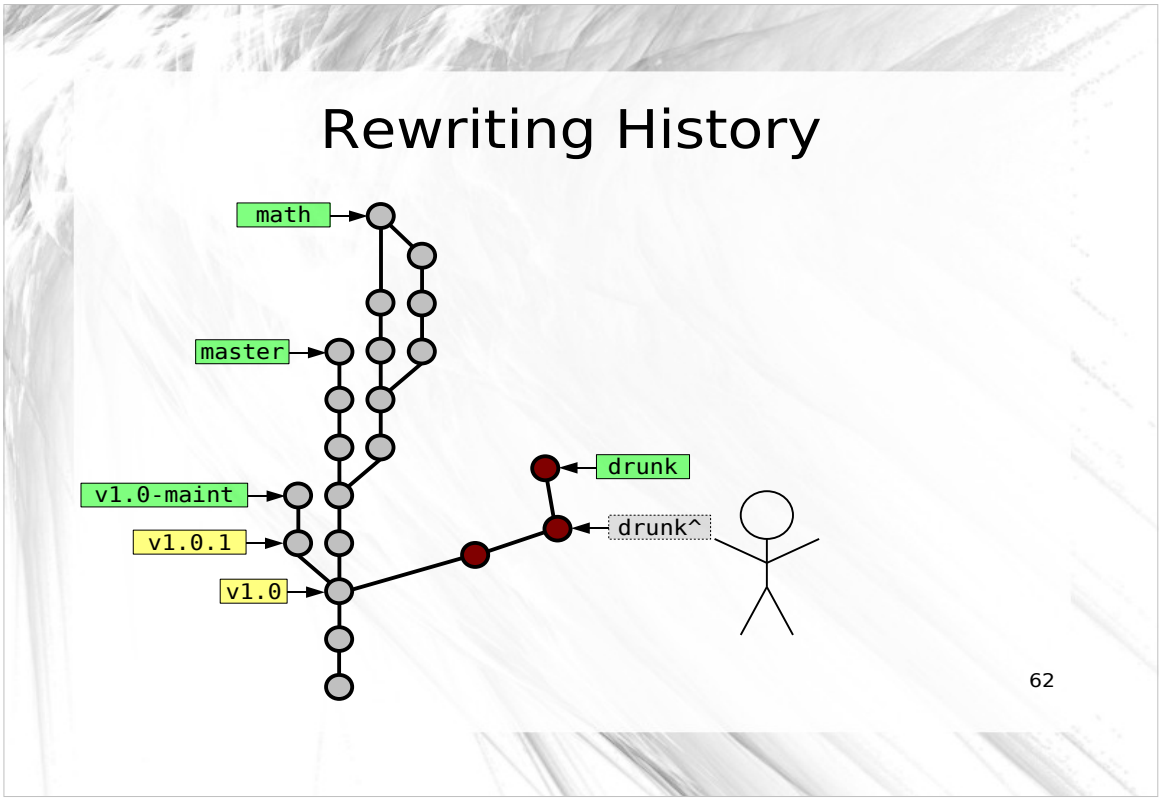
Last night you came home after having a few too many pints of Guinness at the local brewpub and started coding, producing a handful of snapshots along the way. This morning, a review of the code you wrote last night makes you cringe a little bit.

The code is good overall, but you made a lot of mistakes early on that you corrected in later snapshots.

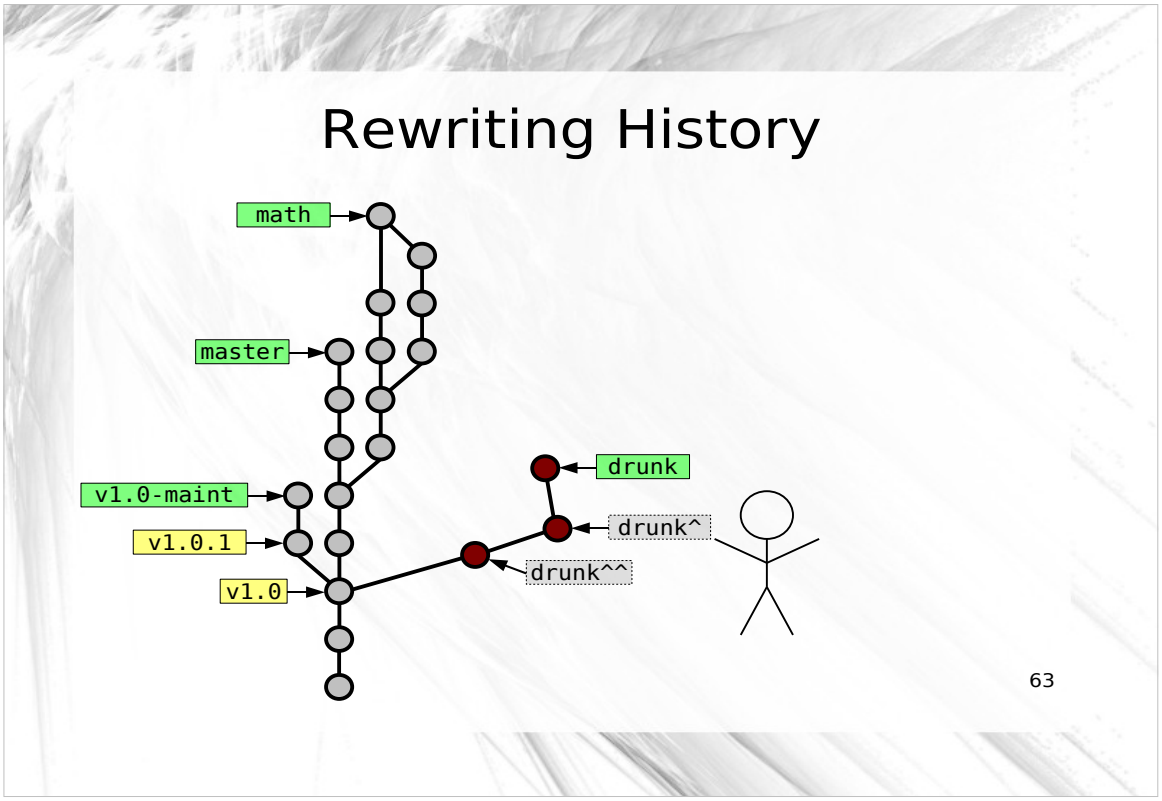



Let’s say the branch on which you did your drunken development is called ‘drunk’ and you made three snapshots after you got home from the bar.

If the name ‘drunk’ points at the latest snapshot on that branch, then we can invent a useful notation to refer to the parent of that snapshot.

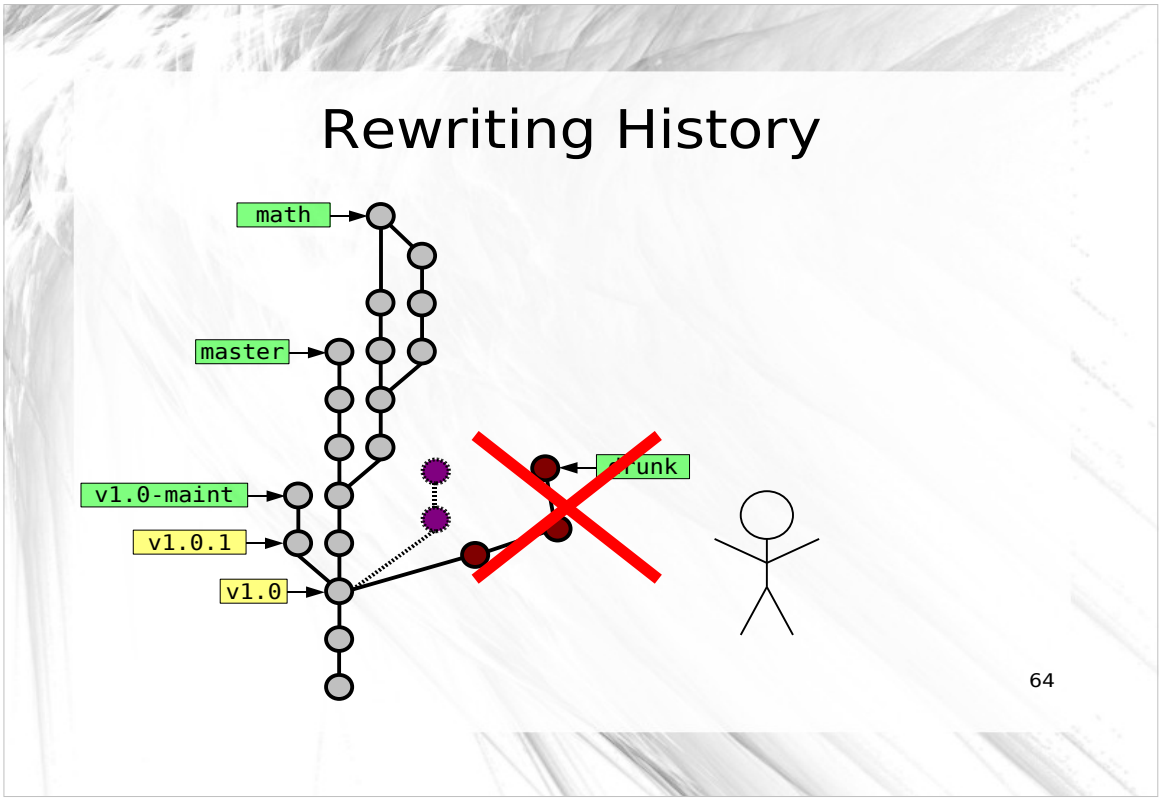



The notation ‘drunk^’ (caret) means the parent of the snapshot pointed to by the branch name ‘drunk’.

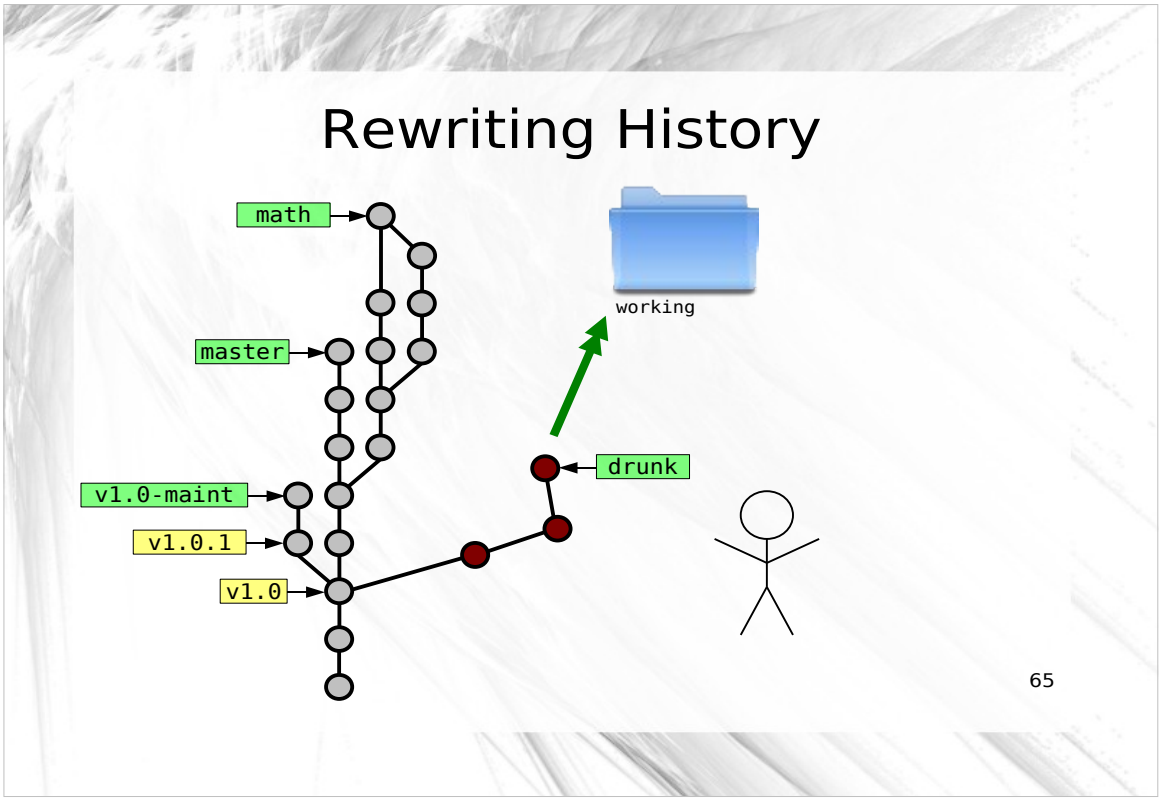



Similarly ‘drunk^^’ means the grandparent of the ‘drunk’ snapshot.

So the three snapshots in chronological order are ‘drunk^^’, ‘drunk^’, and ‘drunk’.

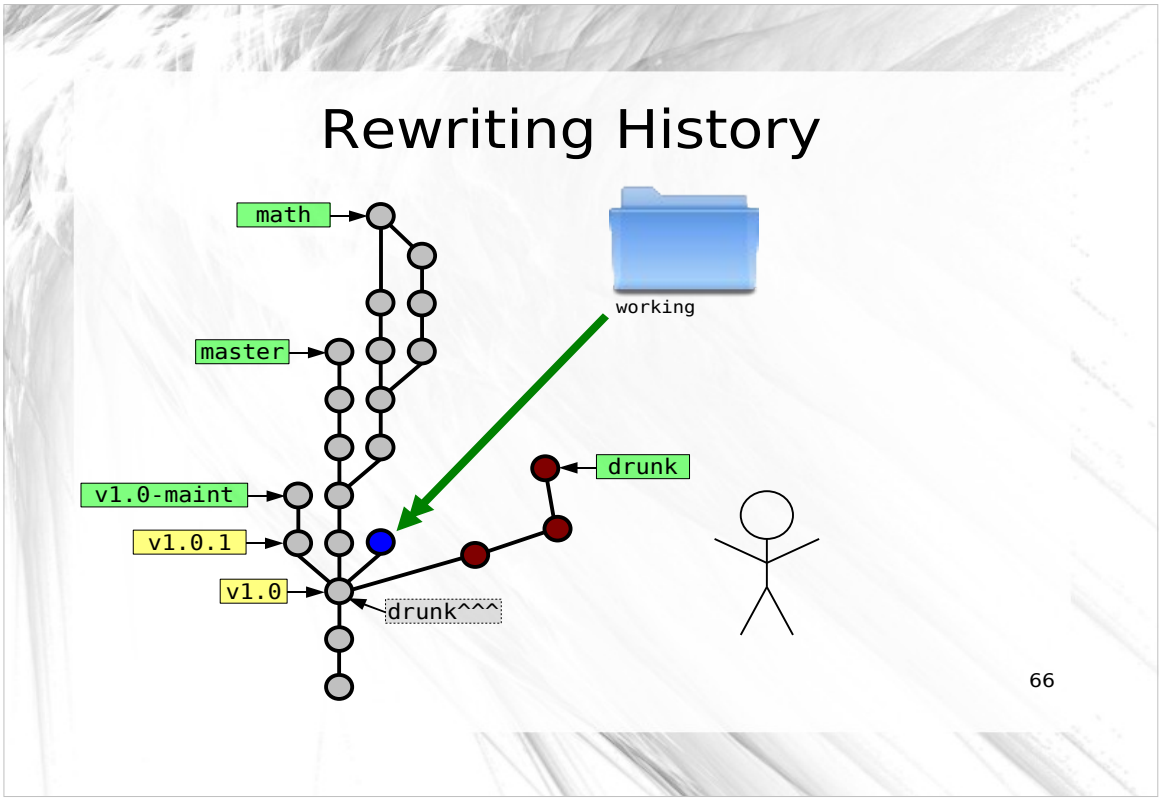



You’d really like those three lousy snapshots to be two clean snapshots. One that changes an existing function, and one that adds a new file.



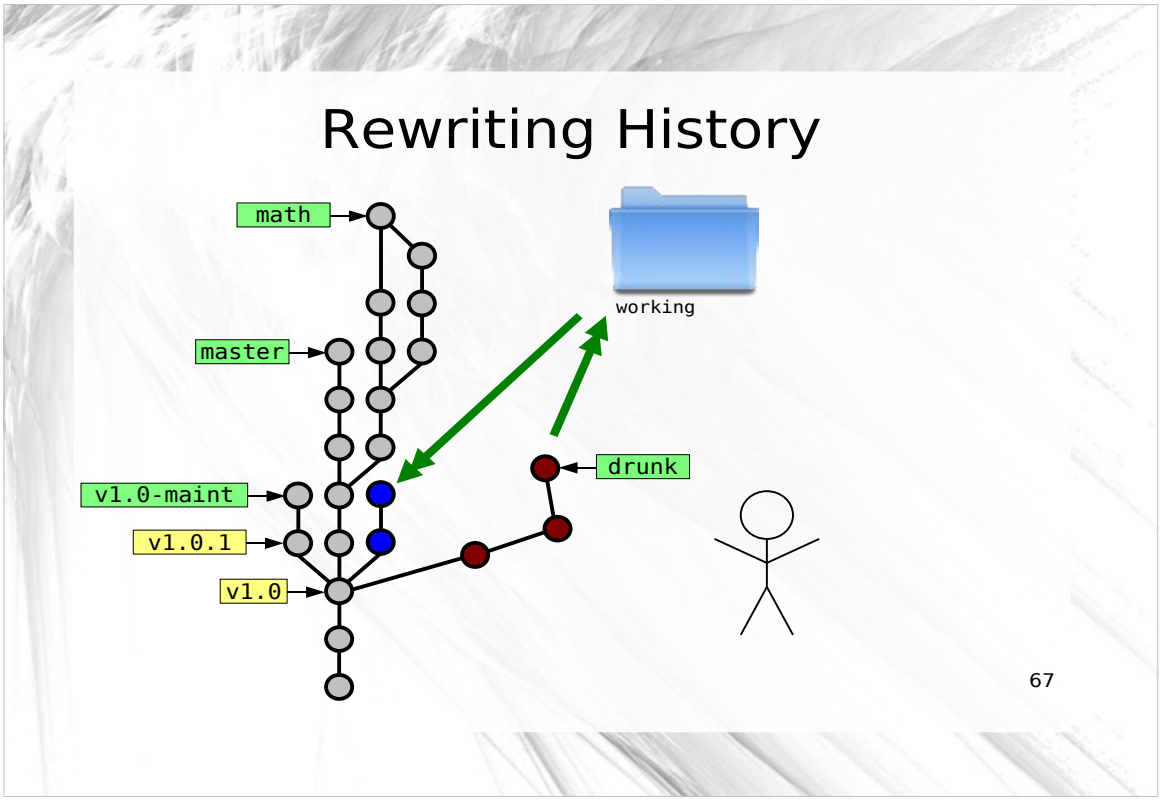

To accomplish this revision of history you copy ‘drunk’ to ‘working’ and delete the file that is new in the series.

Now ‘working’ represents the correct modifications to the existing function.



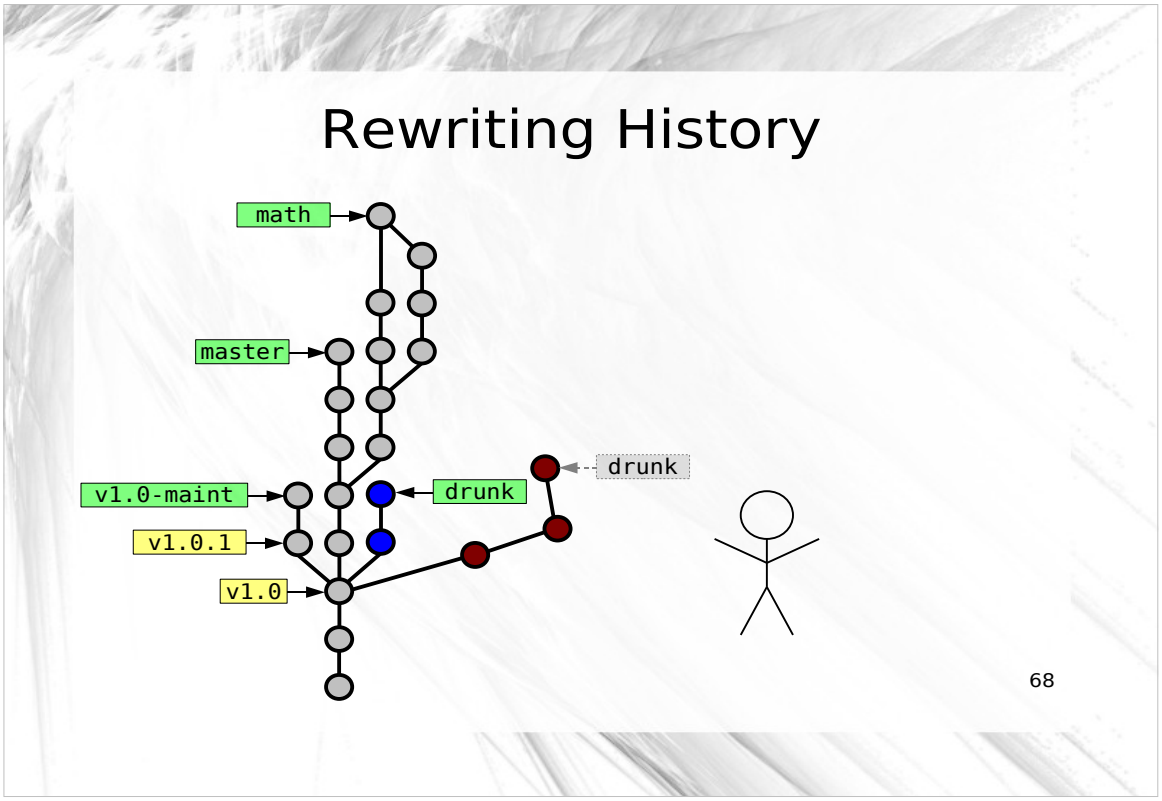

You create a new snapshot from ‘working’ and write the message to be appropriate to the changes.

For the parent you specify the SHA1 of the ‘drunk^^^’ snapshot, essentially creating a new branch off of the same snapshot as last night.

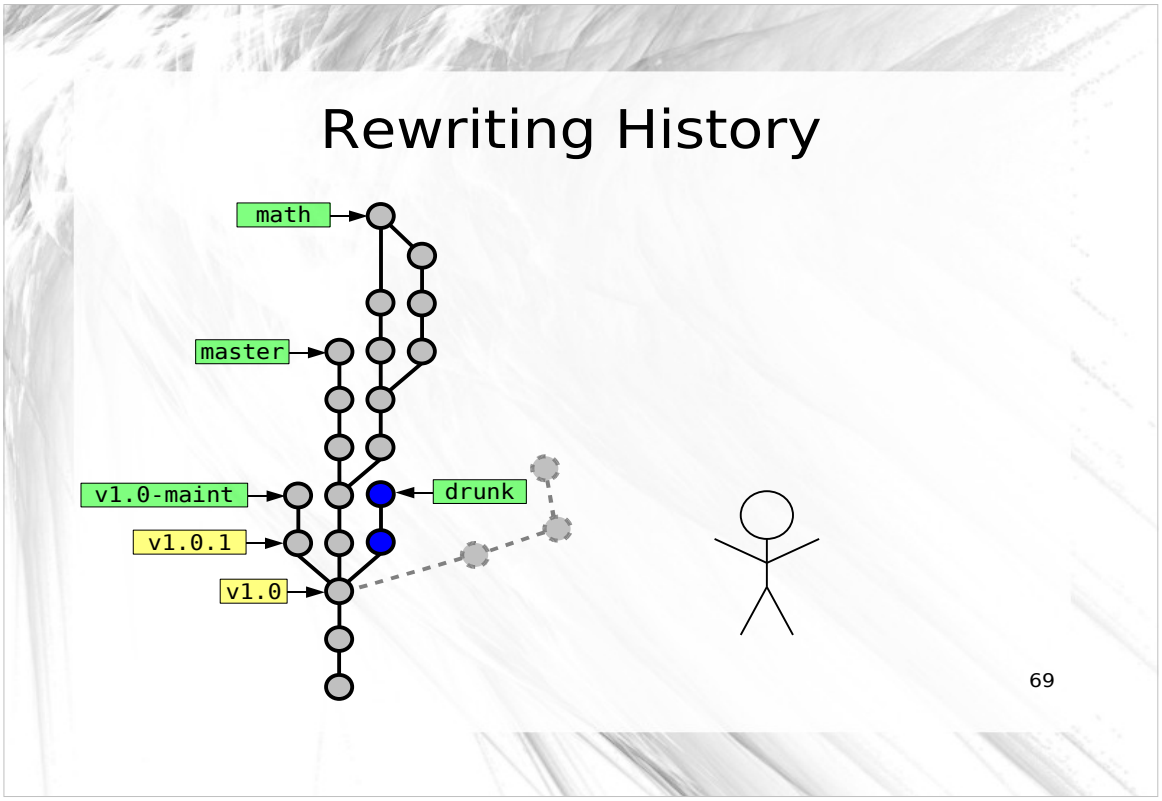



Now you can copy ‘drunk’ to ‘working’ and roll a snapshot with the new file addition.

As the parent you specify that snapshot you created just before this one.



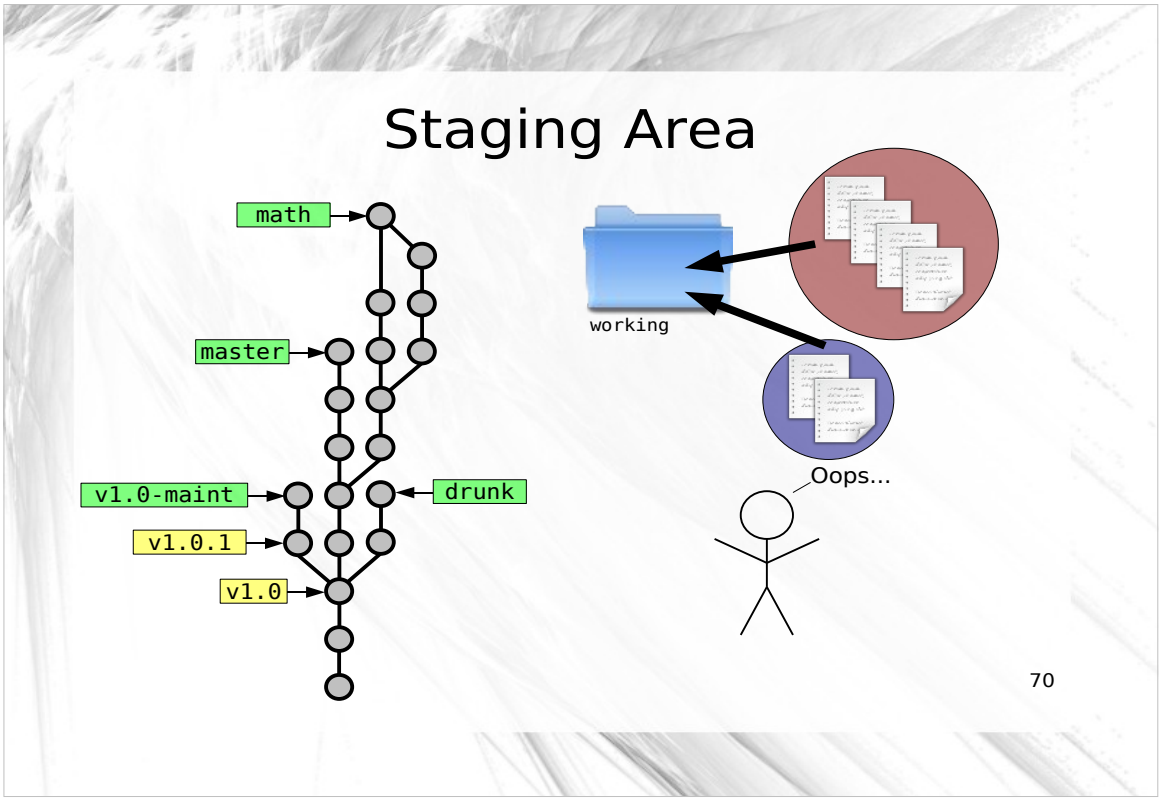

As the last step, you change the branch name ‘drunk’ to point to the last snapshot you just made.




The history of the ‘drunk’ branch now represents a nicer version of what you did last night.

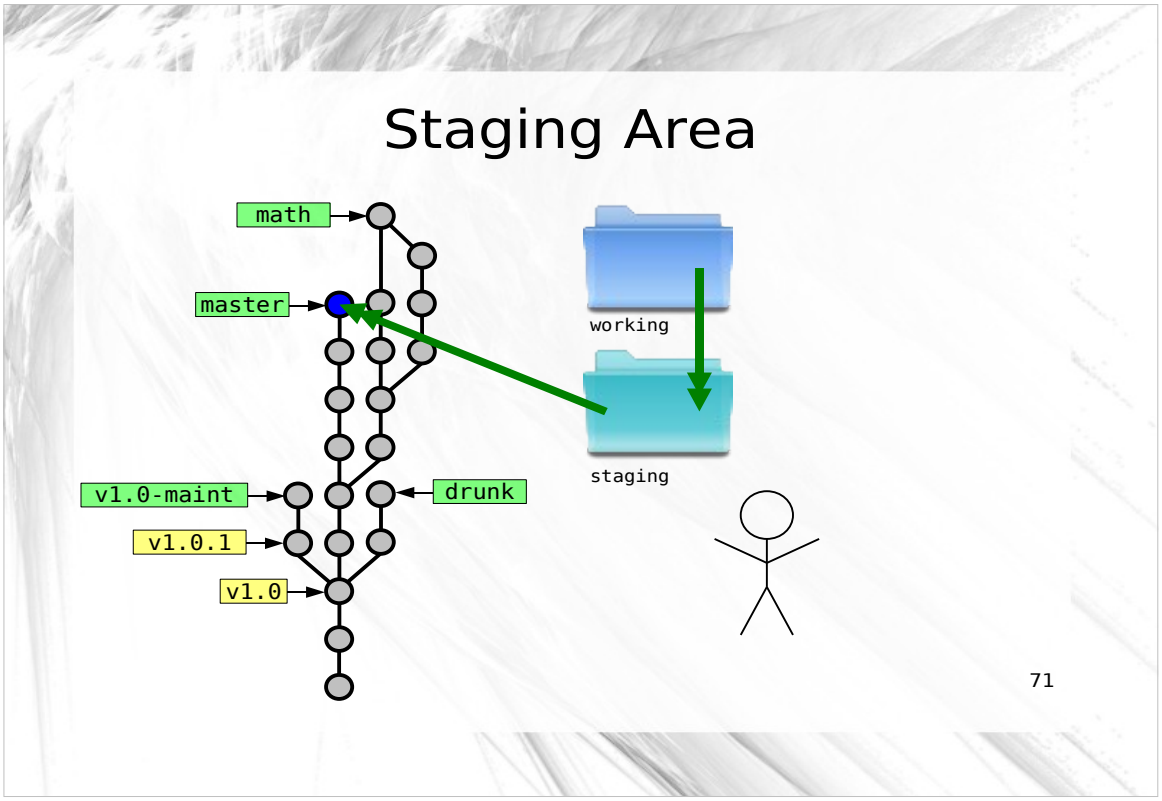
The other snapshots that you’ve replaced are no longer needed so you can delete them or just leave them around for posterity.

No branch names are currently pointing at them so it will be hard to find them later on, but if you don’t delete them, they’ll stick around.



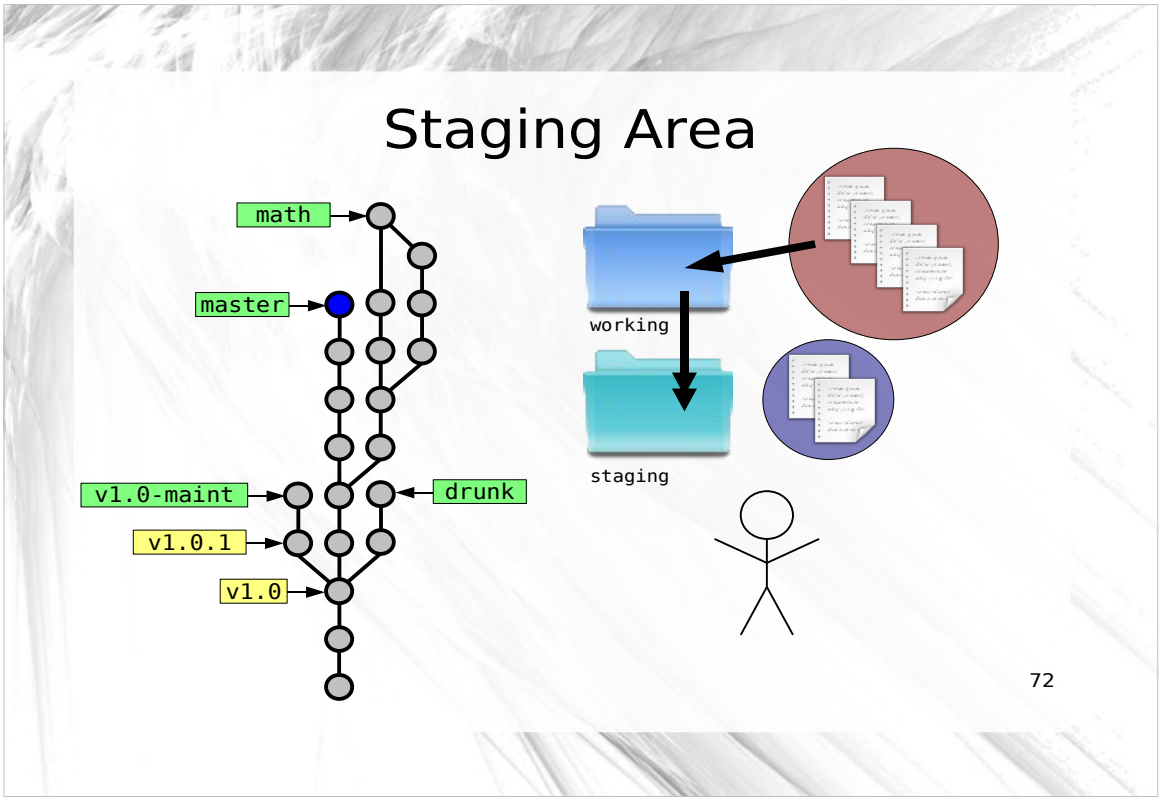

As much as you try to keep your new modifications related to a single feature or logical chunk, you sometimes get sidetracked and start hacking on something totally unrelated.

Only half-way into this do you realize that your working directory now contains what should really be separated as two discrete snapshots.

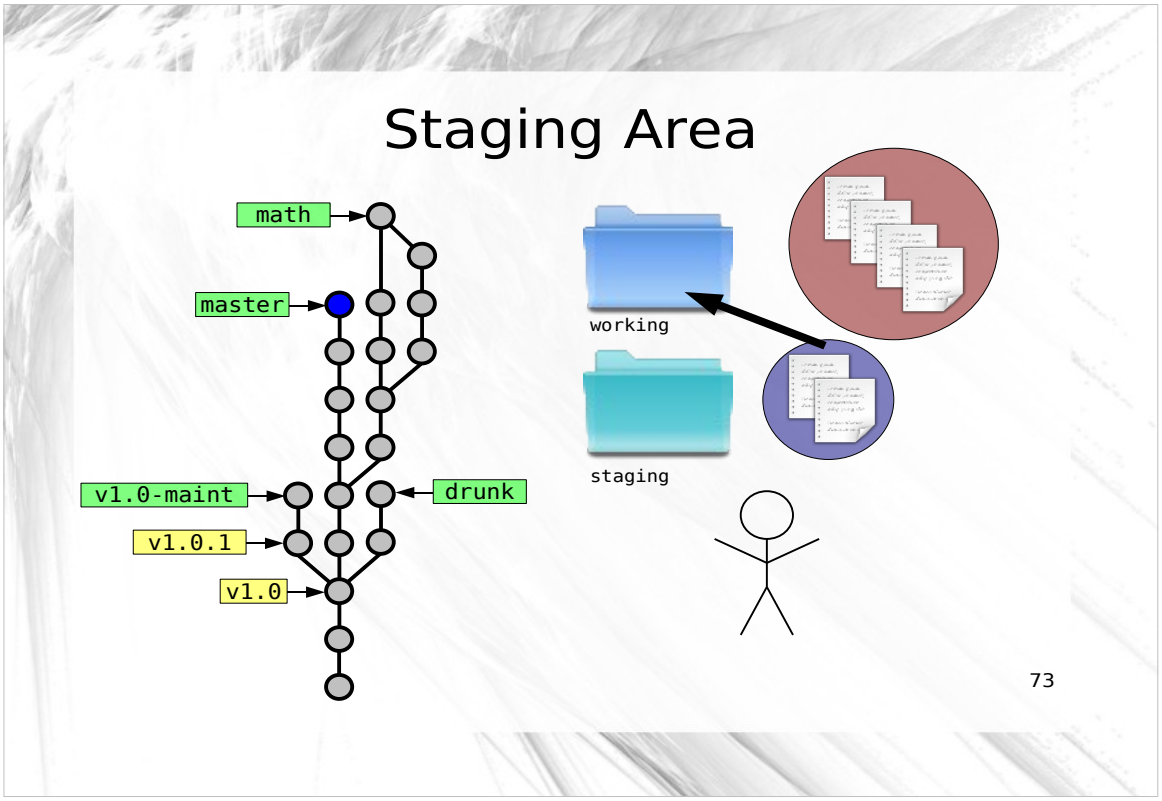



To help you with this annoying situation, the concept of a “staging” directory is useful. This area acts as an intermediate step between your working directory and a final snapshot.

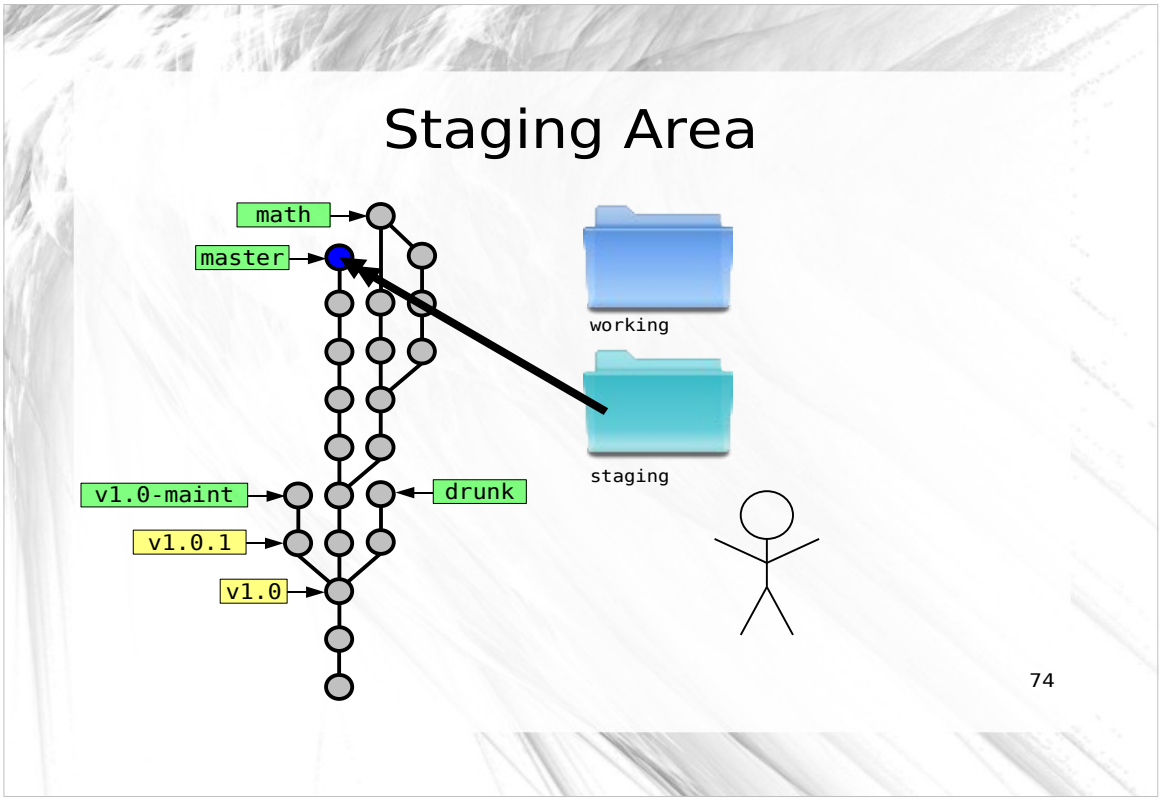
Each time you finish a snapshot, you first copy it to the “staging” directory, and and then create the snapshot from the “staging” directory.




If it belongs, you mimic the change inside “staging”.

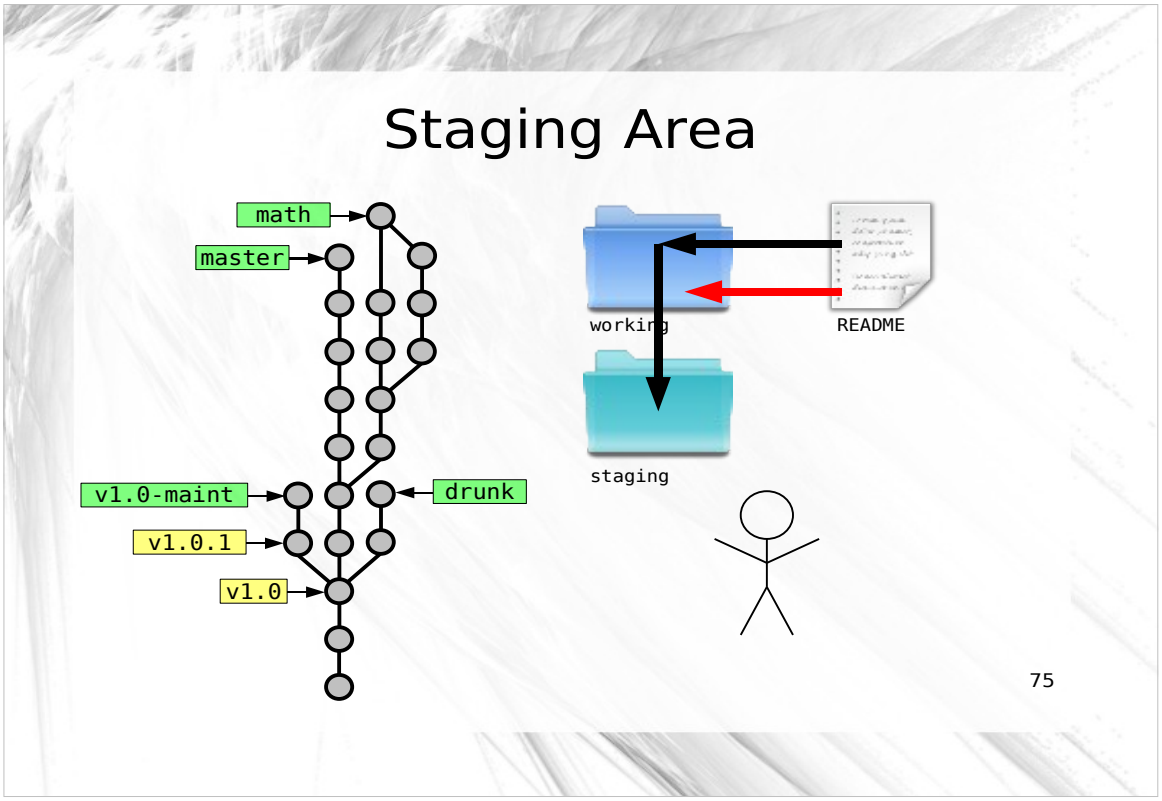



If it doesn’t, you can leave it in “working” and make it part of a later snapshot.




When you're satisfied with the state of the “staging” directory, you create a new snapshot from it.

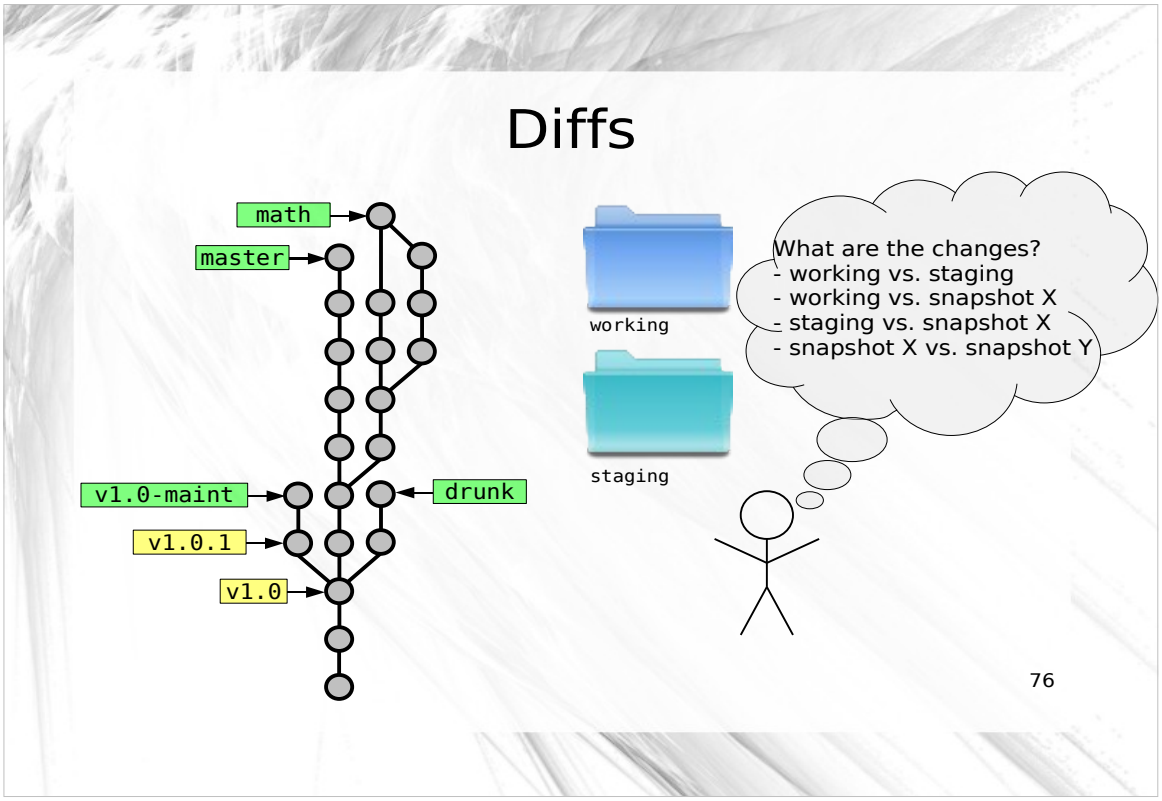
This separation of coding and preparing the stage makes it easy to specify what is and is not included in the next snapshot. You no longer have to worry too much about making an accidental, unrelated change in your working directory.




You have to be a bit careful, though. Consider a file named “README”. You make an edit to this file and then mimic that in “staging”. You go on about your business, editing other files. After a bit, you make another change to “README”.

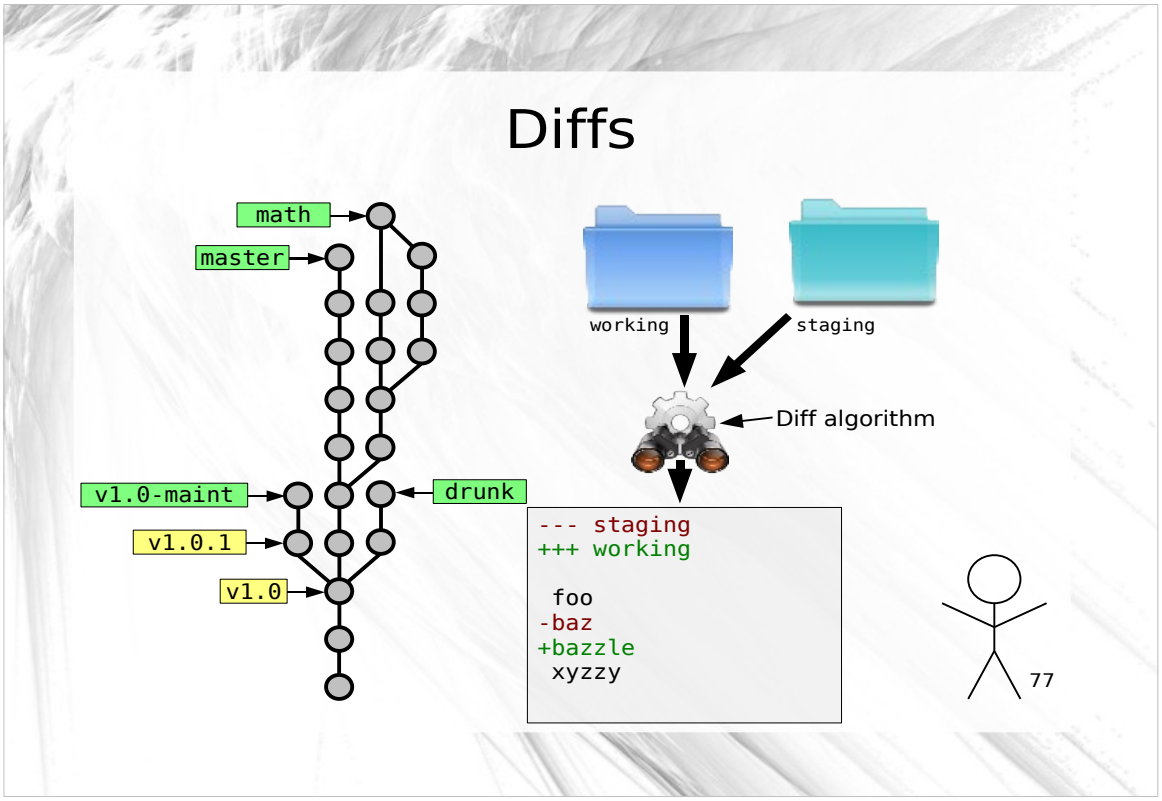
Now you have made two changes to that file, but only one is in the staging area! Were you to create a snapshot now, your second change would be absent.

The lesson is this: every new edit must be added to the staging area if it is to be part of the next snapshot.



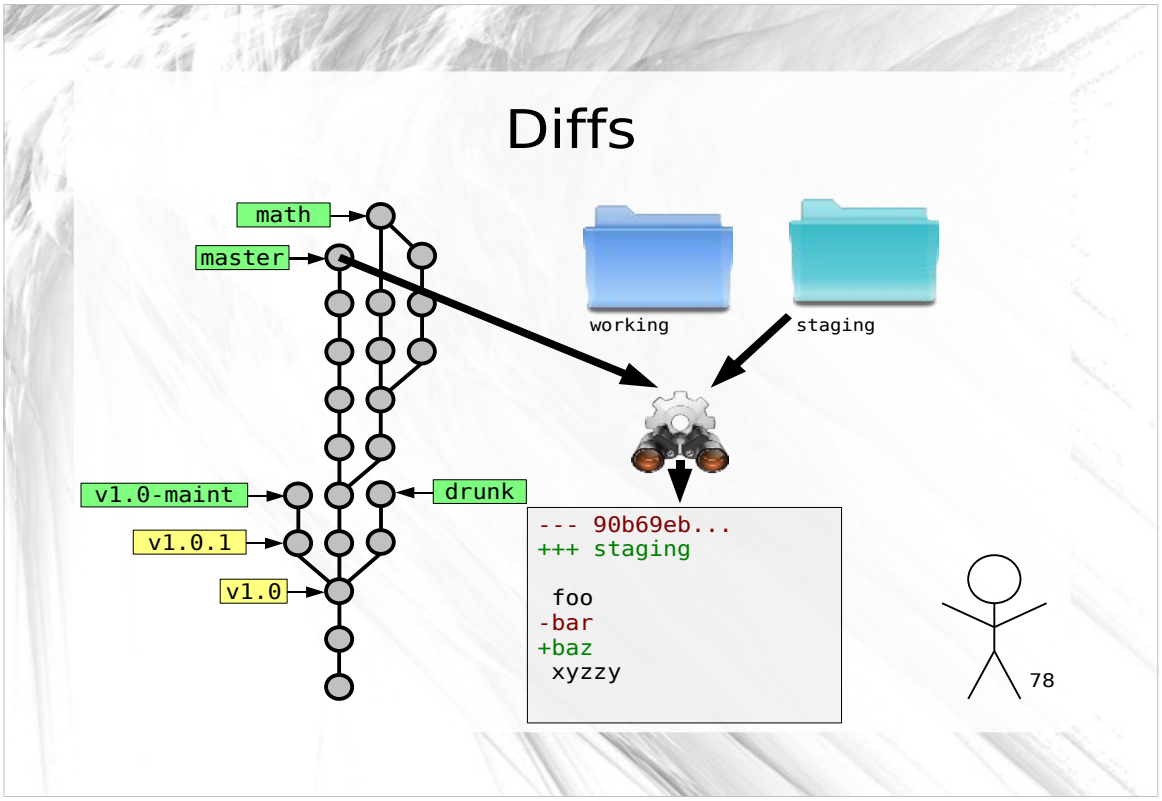

With a working directory, a staging area, and loads of snapshots laying around, it starts to get confusing as to what the specific code changes are between these directories.

A snapshot message only gives you a summary of what changed, not exactly what lines were changed between two files.

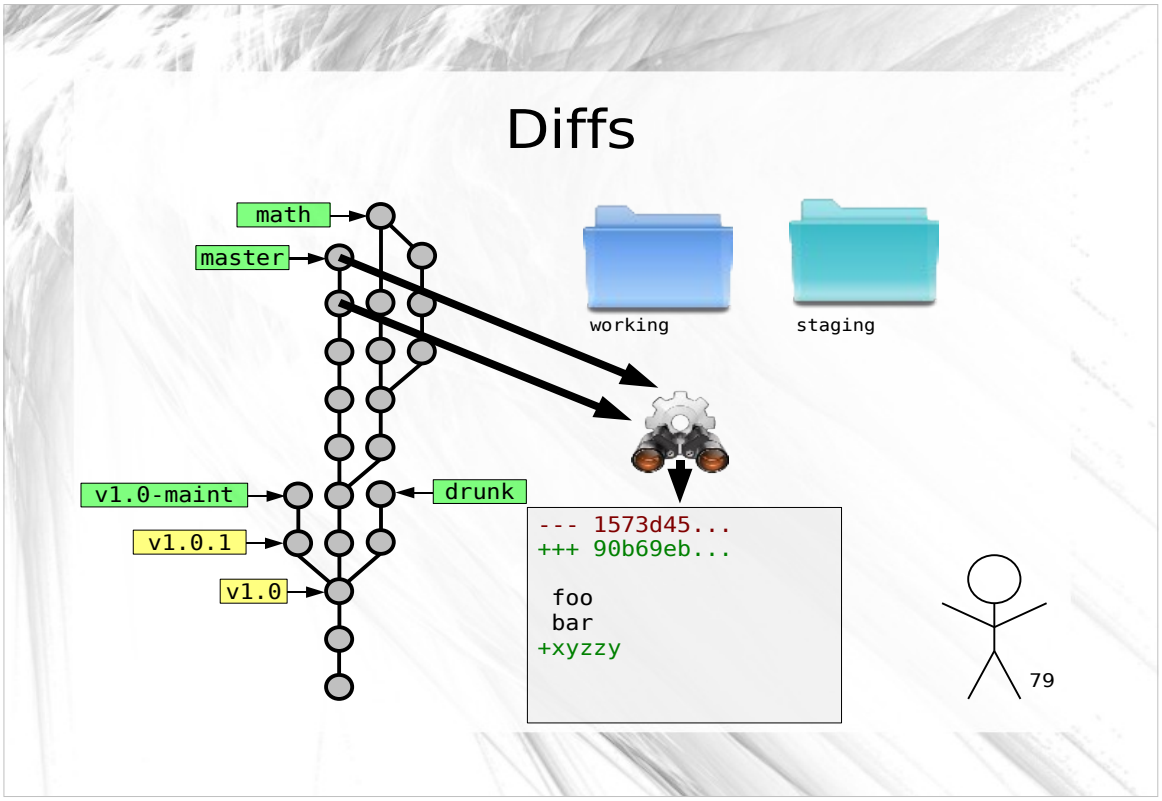



Using a diffing algorithm, you can implement a small program that shows you the differences in two codebases.

As you develop and copy things from your working directory to the staging area, you'll want to easily see what is different between the two, so that you can determine what else needs to be staged.

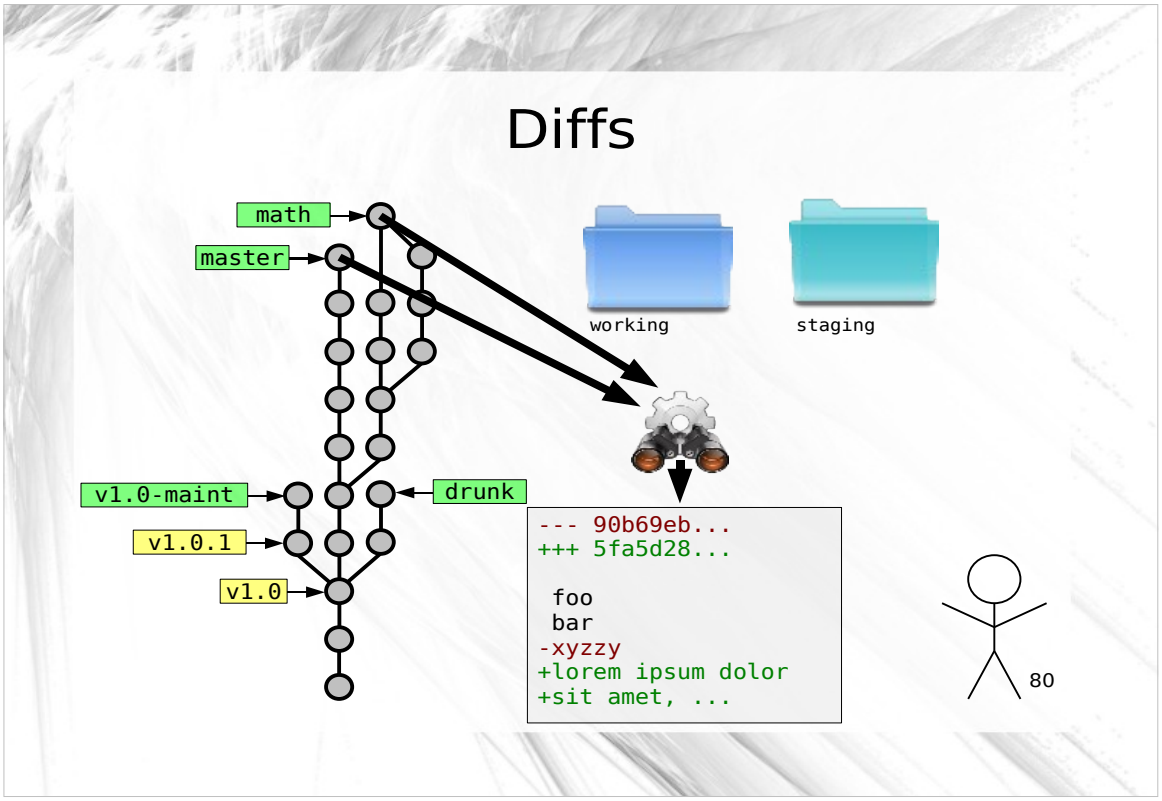



It’s also important to see how the staging area is different from the last snapshot, since these changes are what will become part of the next snapshot you produce.

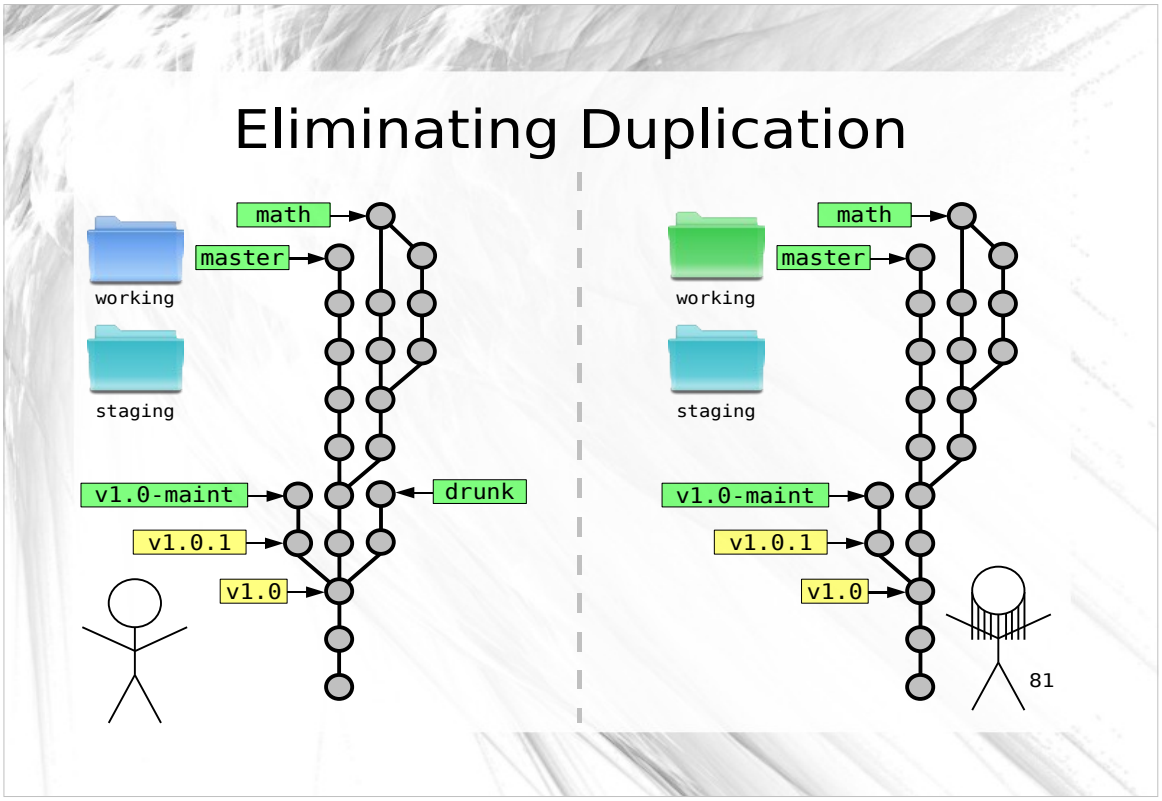



There are many other diffs you might want to see.

The differences between a specific snapshot and its parent would show you the “changeset” that was introduced by that snapshot.



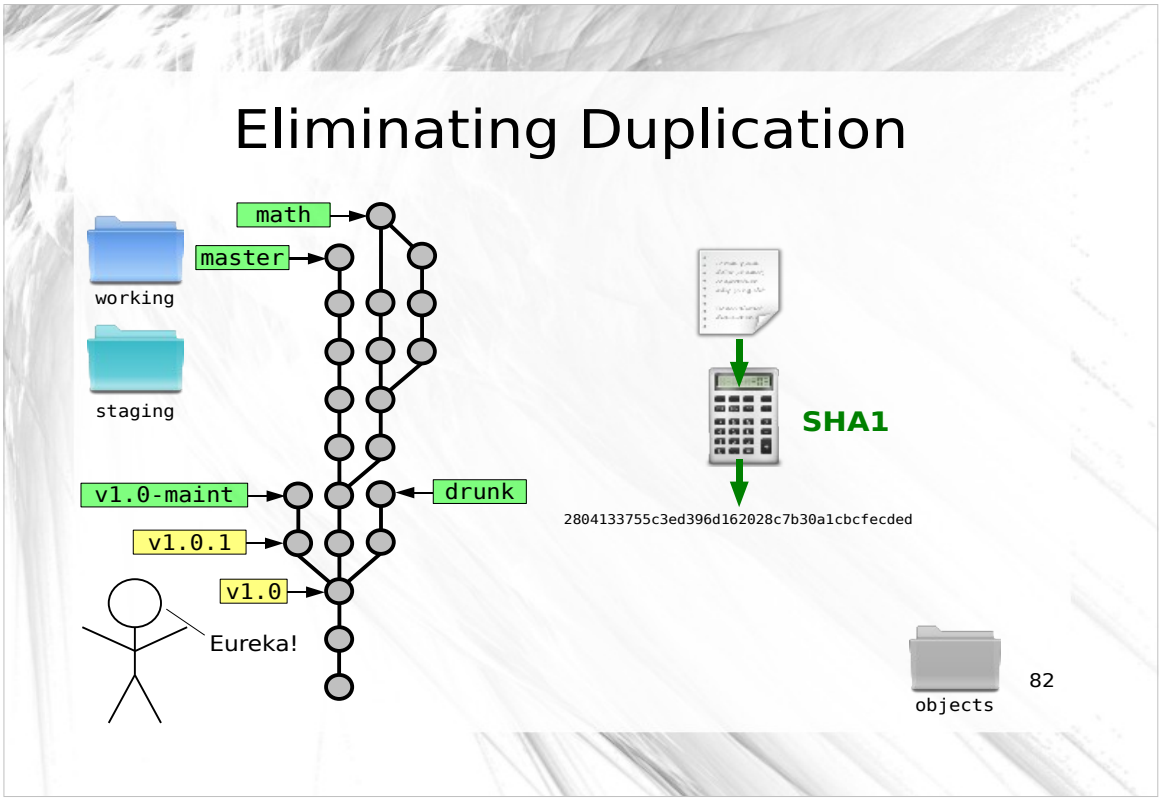

The diff between two branches would be helpful for making sure your development doesn’t wander too far away from the mainline.




Remember Zoe?

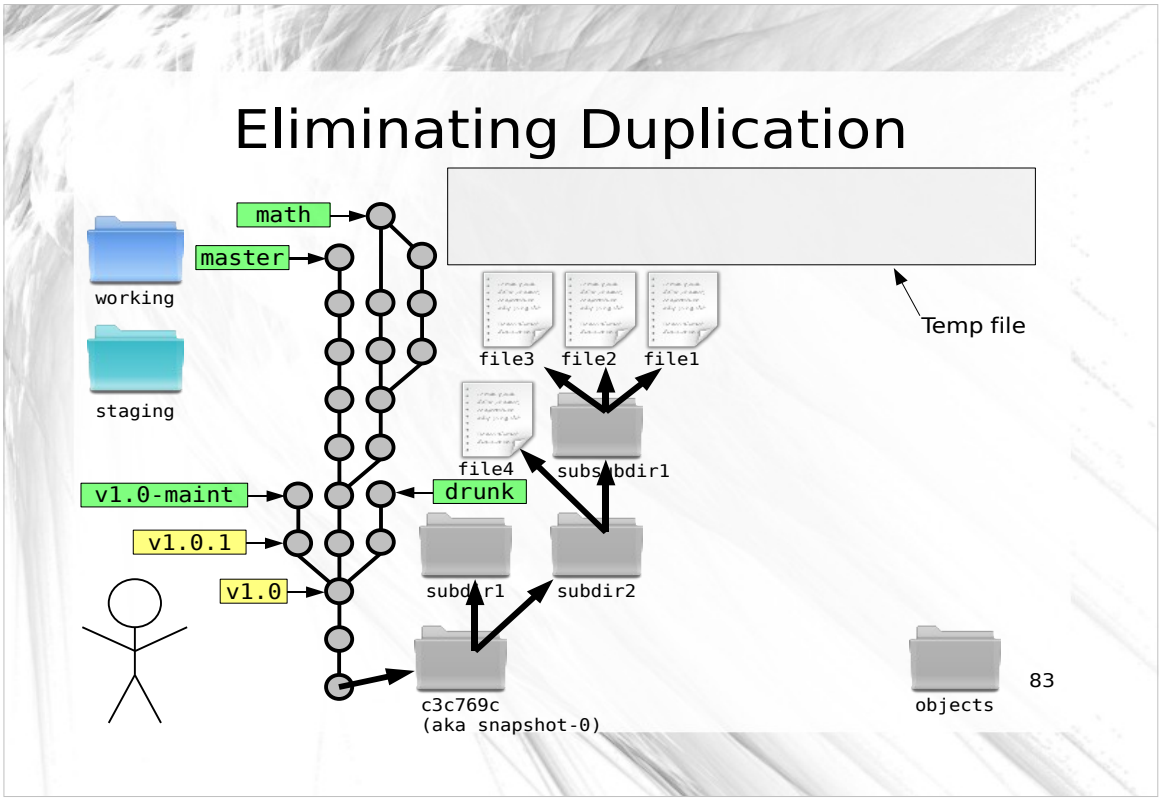
After a few more trips to Namibia, Istanbul, and Galapagos, Zoe starts to complain that her hard drive is filling up with hundreds of nearly identical copies of the software.

You too have been feeling like all the file duplication is wasteful. After a bit of thinking, you come up with something very clever.




You remember that the SHA1 hash produces a short string that is unique for a given file contents.

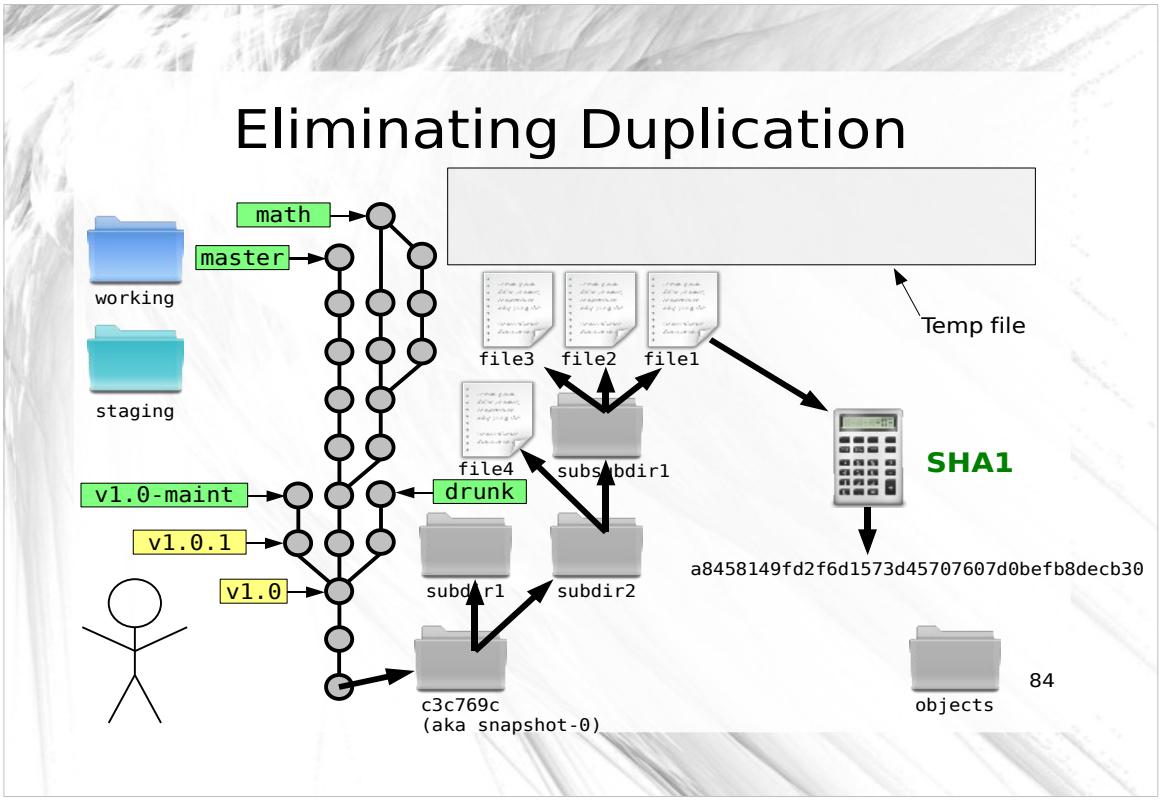
Starting with the very first snapshot in the project history, you start a conversion process. First, you create a directory named “objects” outside of the code history.



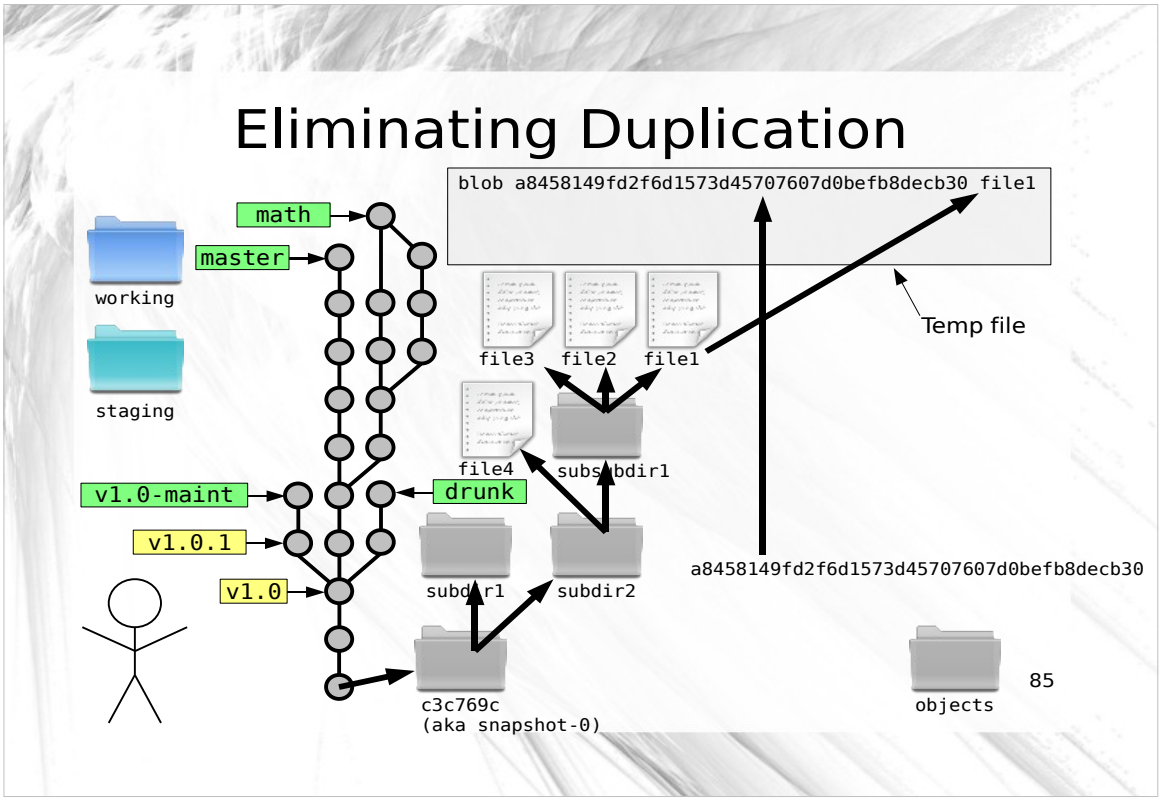

Next, you find the most deeply nested directory in the snapshot.

Additionally, you open up a temporary file for writing.

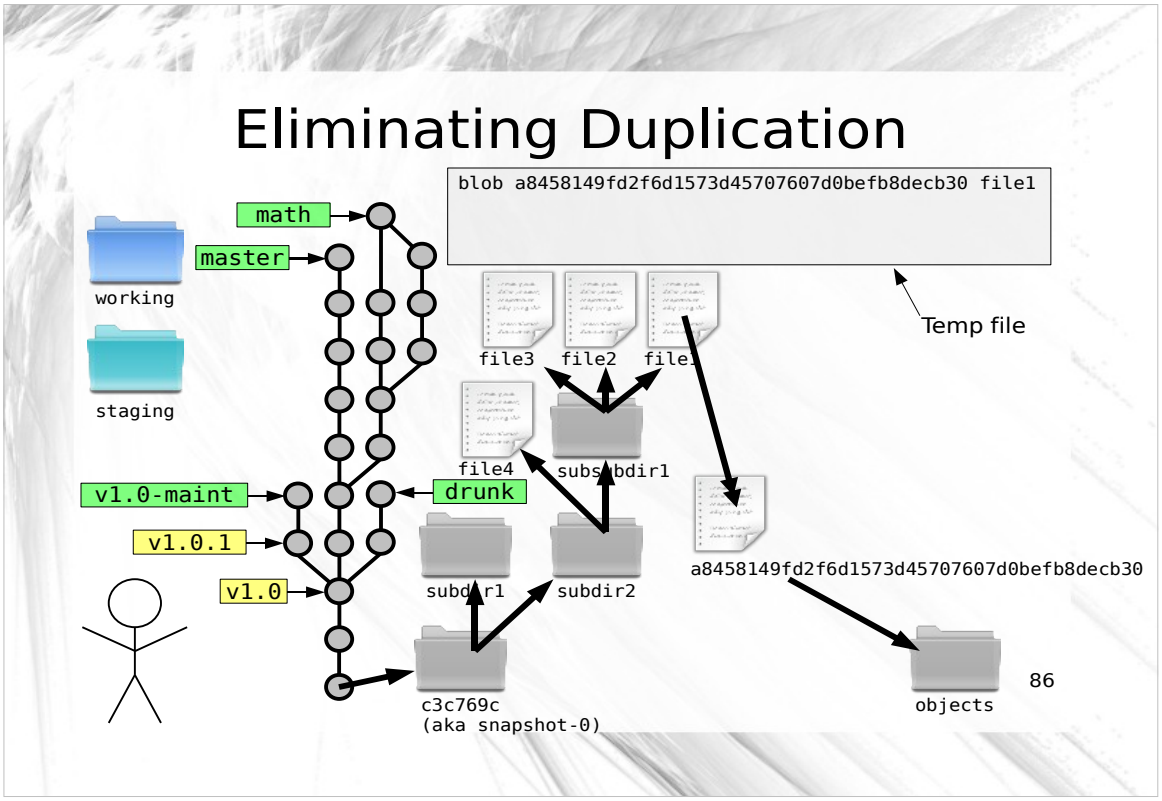
For each file in the most deeply nested directory, you perform three steps:



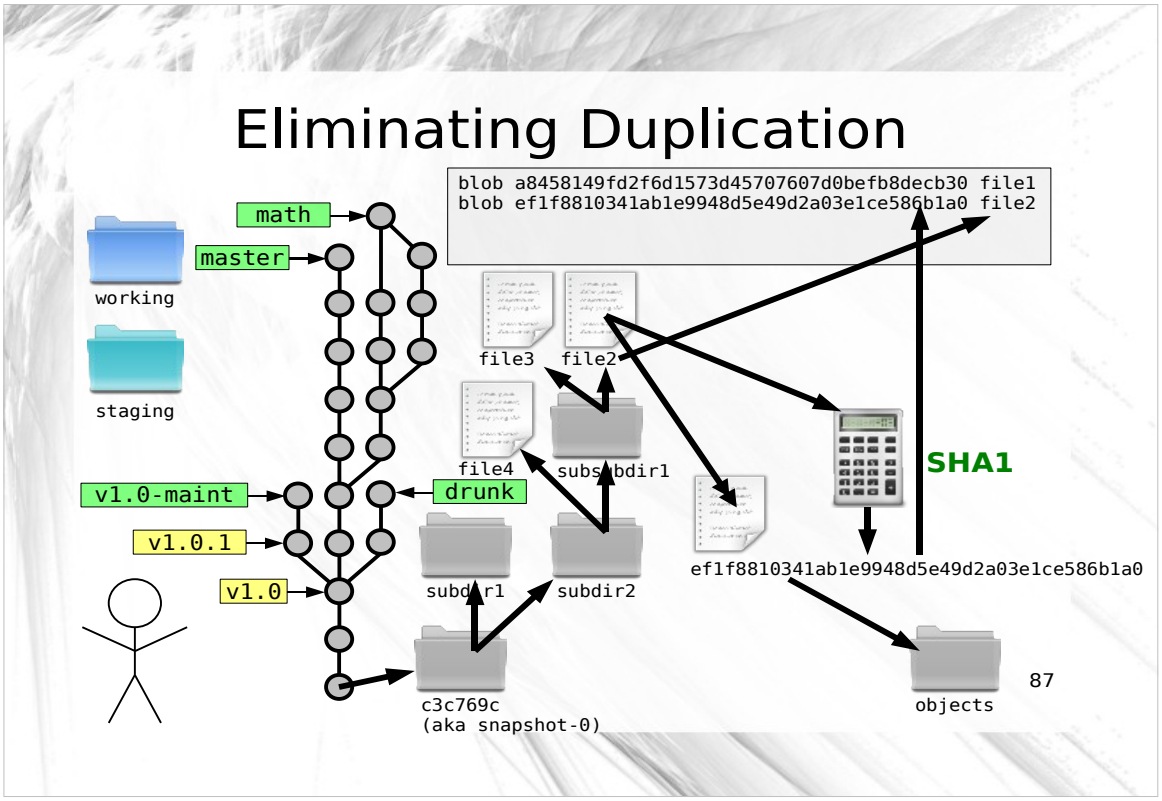

Step 1: Calculate the SHA1 of the contents.



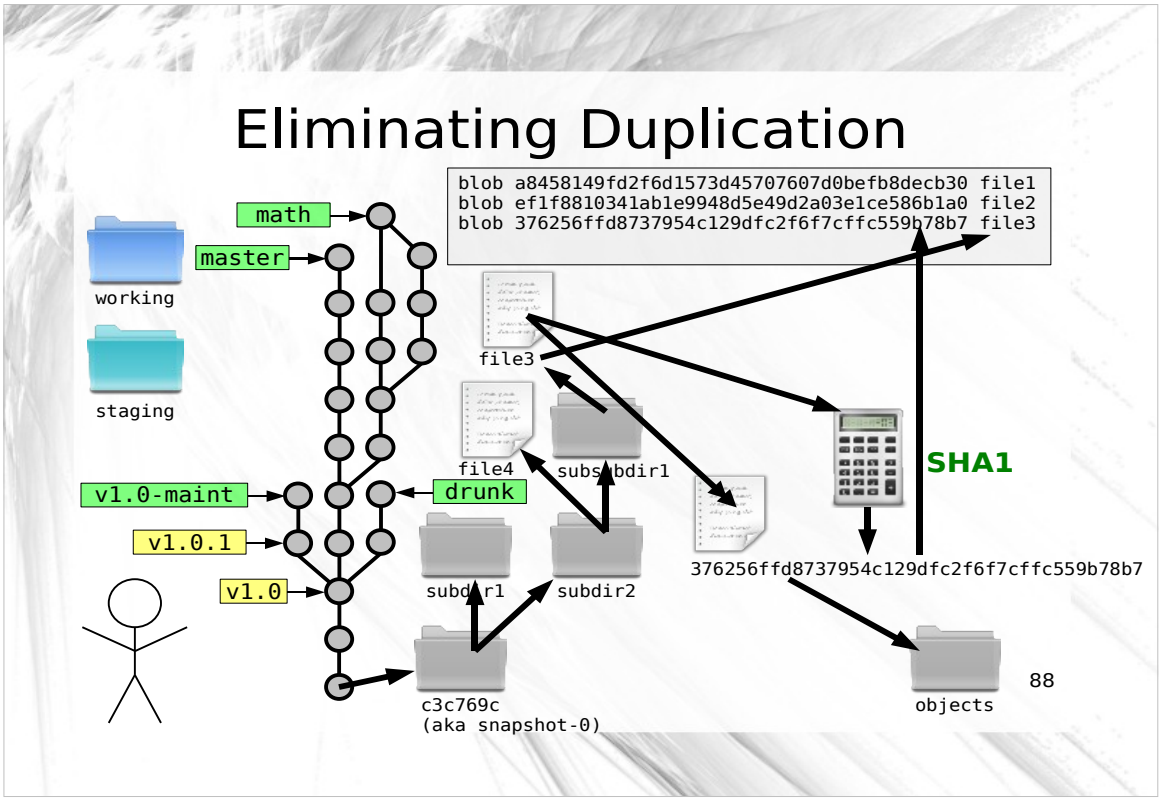

Step 2: Add an entry into the temp file that contains the word ‘blob’ (binary large object), the SHA1 from the first step, and the filename.



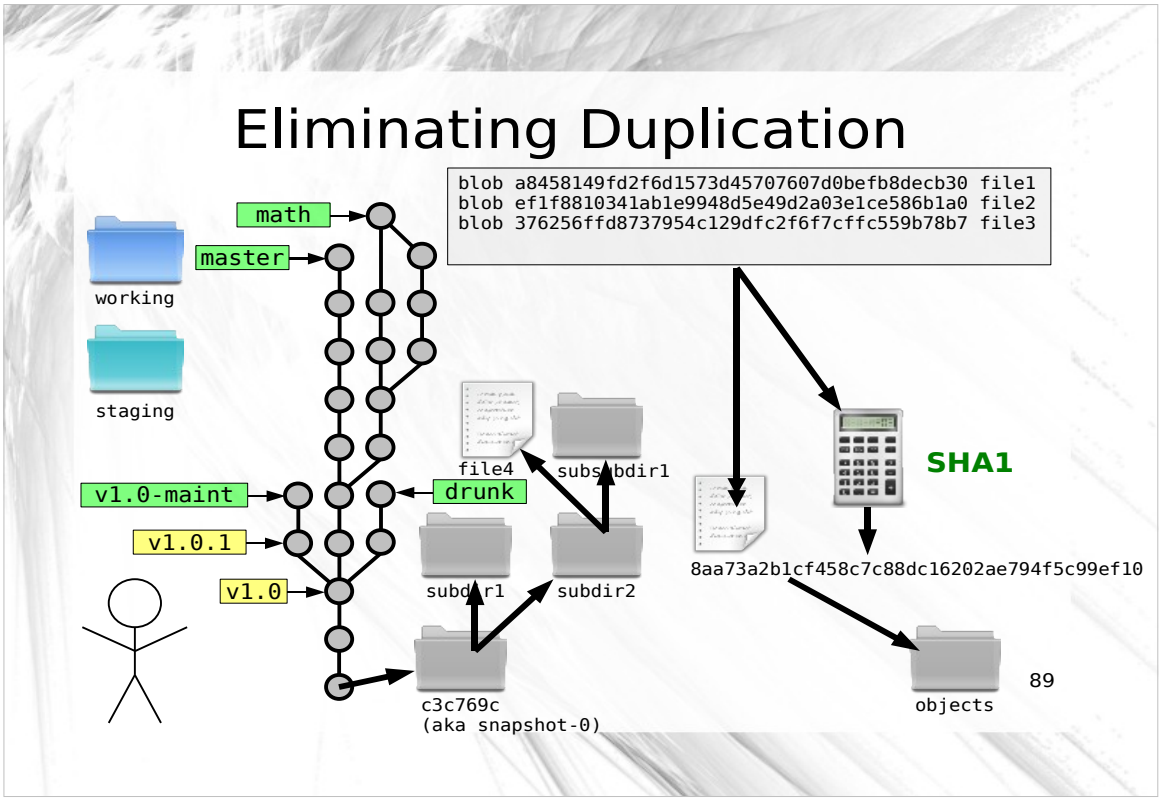

Step 3: Copy the file to the objects directory and rename it to the SHA1 from step 1.




Repeat this process for the other two files in that directory.

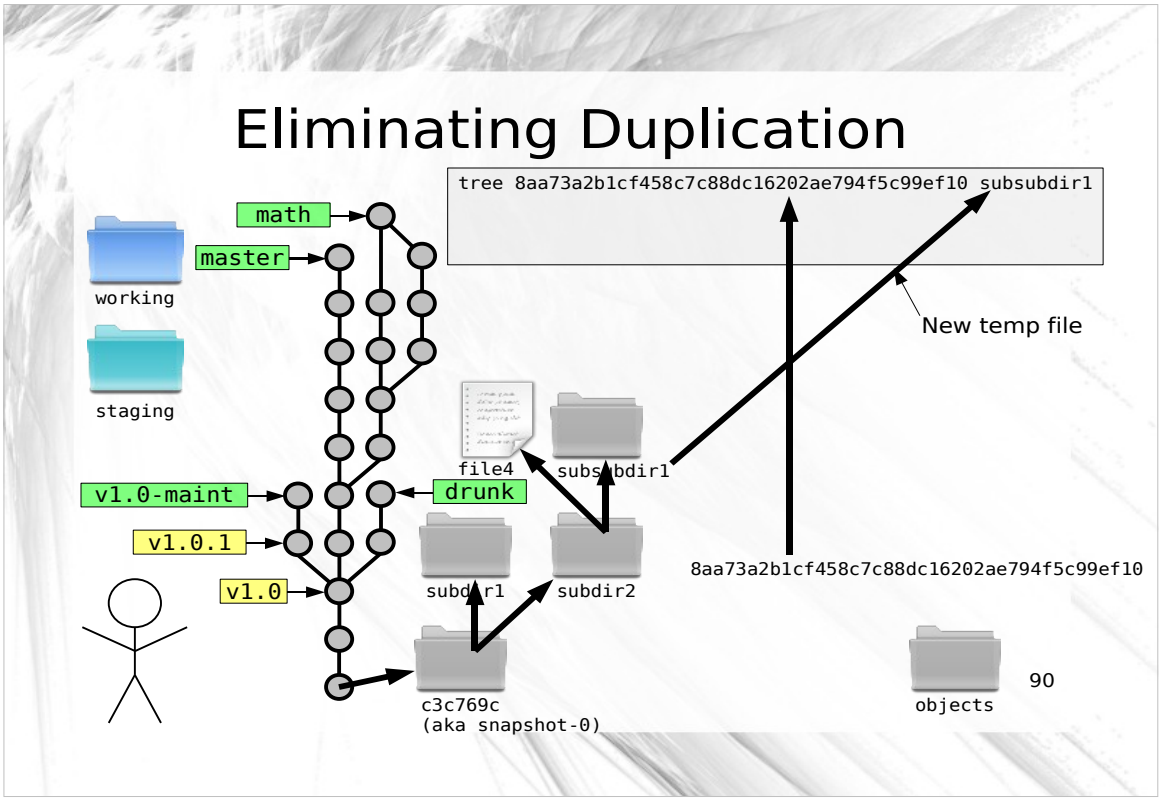

Eliminating Duplication
blob a8458149fd2f6d1573d45707607d0befb8decb30 file1
blob ef1f8810341ab1e9948d5e49d2a03e1ce586b1a0 file2
blob 376256ffd8737954c129dfc2f6f7cffc559b78b7 file3
math
master
working
staging
v1.0-maint
drunk
v1.0.1
v1.0
file3
file4
subsubdir1
subdir1
subdir2
SHA1
376256ffd8737954c129dfc2f6f7cffc559b78b7
c3c769c
(aka snapshot-0)
objects
88

...



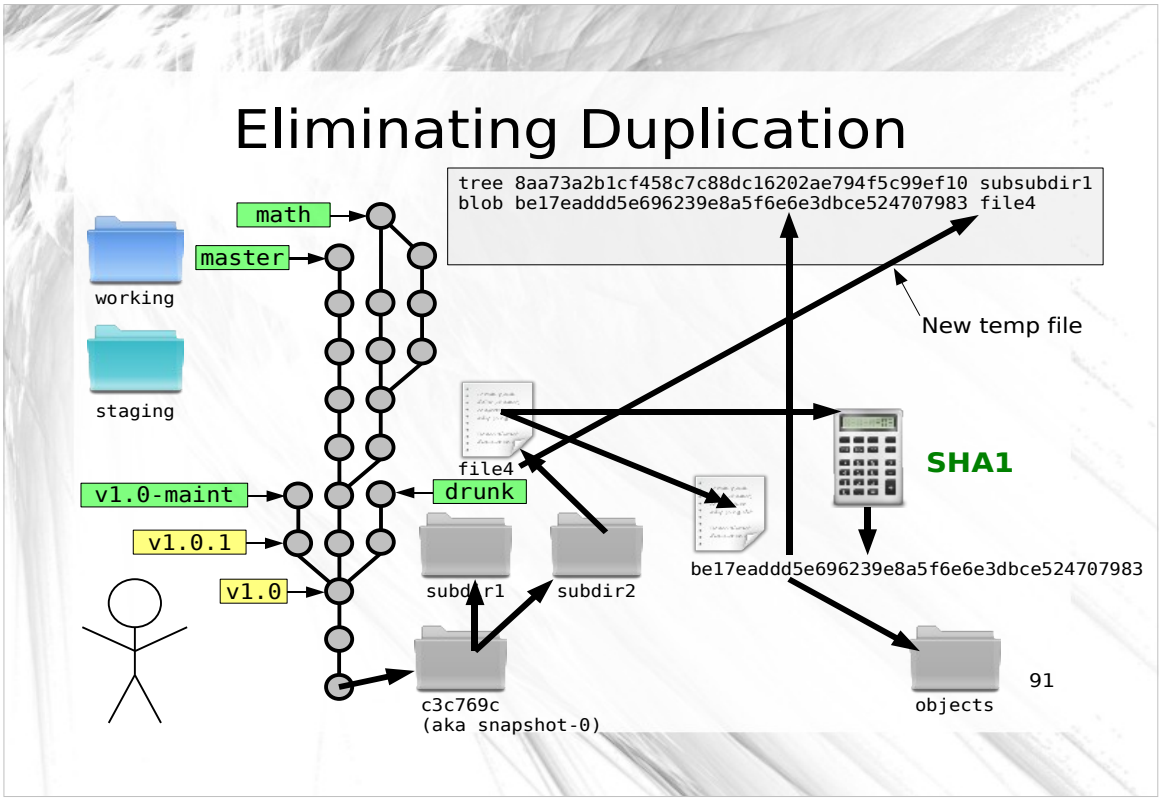

Once finished with all the files, find the SHA1 of the temp file contents and use that to name the temp file, also placing it in the objects directory.

Note that the contents of the temp file exactly mirrors the contents of “subsubdir1”, almost some kind of a directory listing. We use this fact in the next phase:

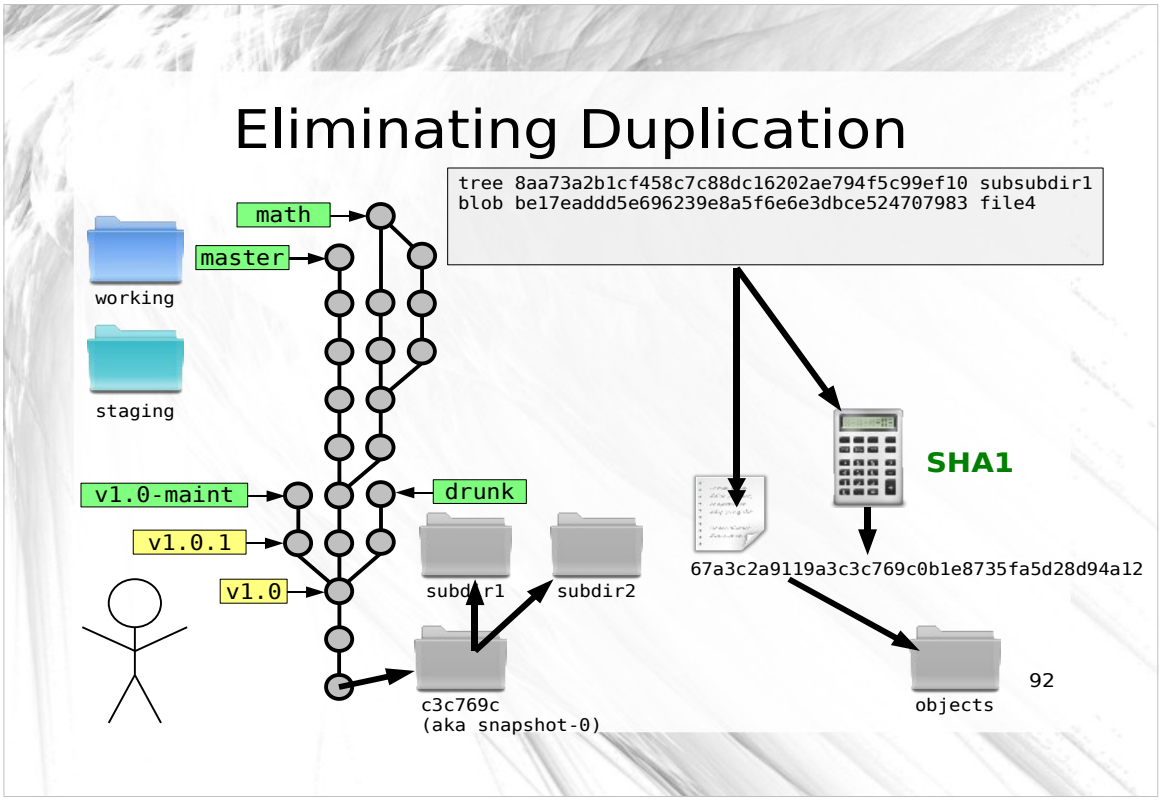



Now, move up one directory and start over. Only this time, when you get to the entry for the directory that you just processed, enter the word ‘tree’, the SHA1 of the temp file from last time, and the directory’s name into the new temp file.

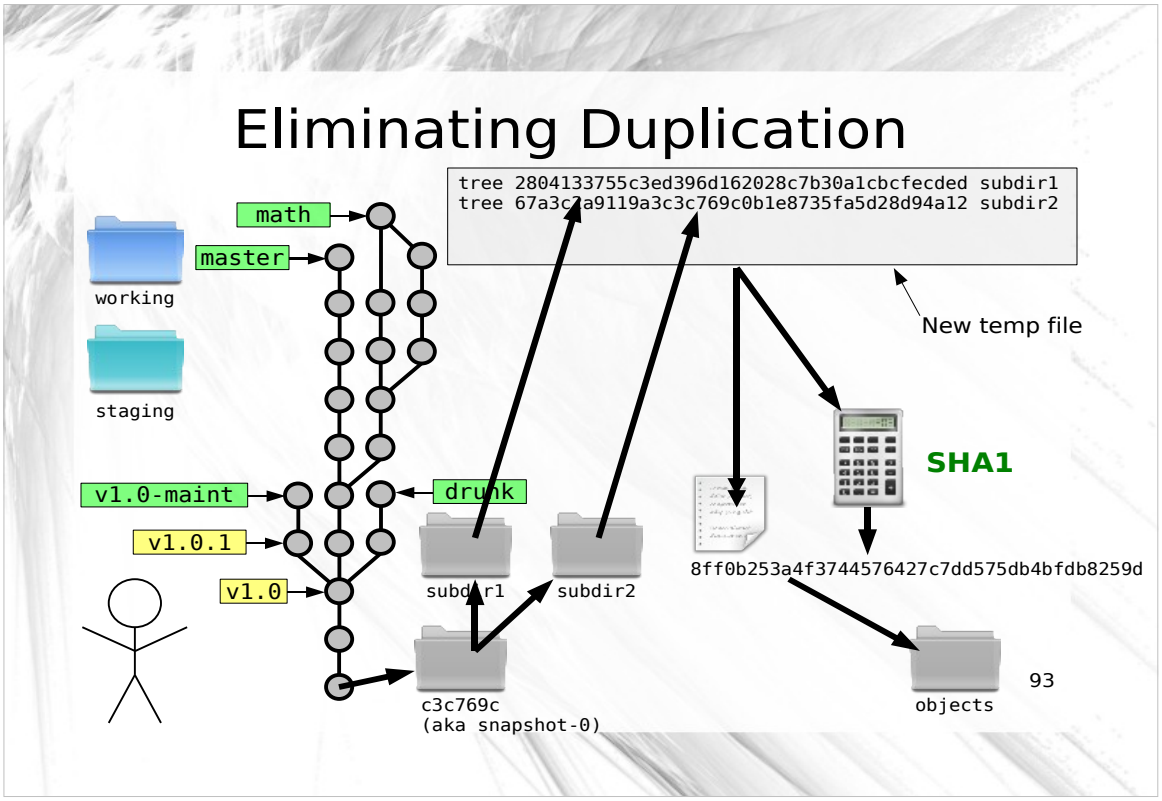
So, to identify the contents of the “subsubdir1” directory, we re-use the SHA1 of the “directory listing” from the last phase.



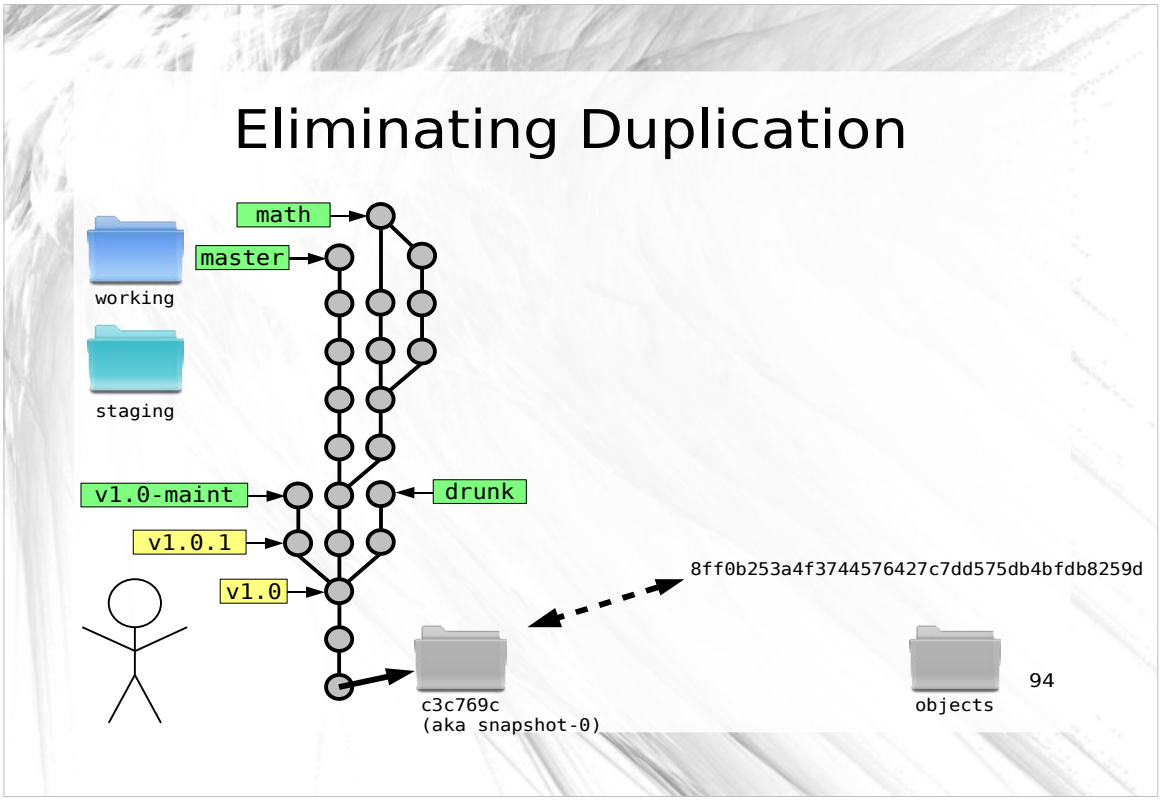

Now, at this level there's also a regular file, “file4”. We handle this in the same way as the previous files in “subsubdir1”.




When we finish that directory level, we hash and store the directory listing, and keep going at the next level.

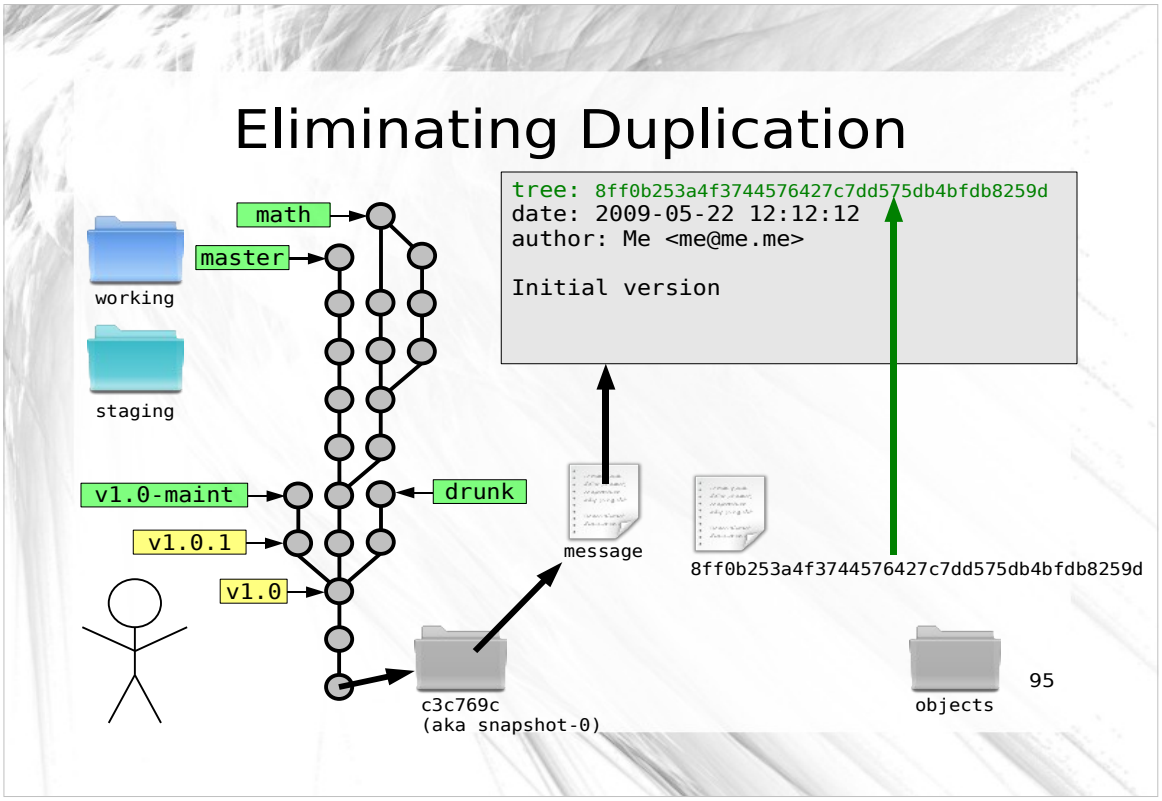



In this fashion you can build up a tree of directory object files that contain the SHA1s and names of the files and directory objects that they contain.



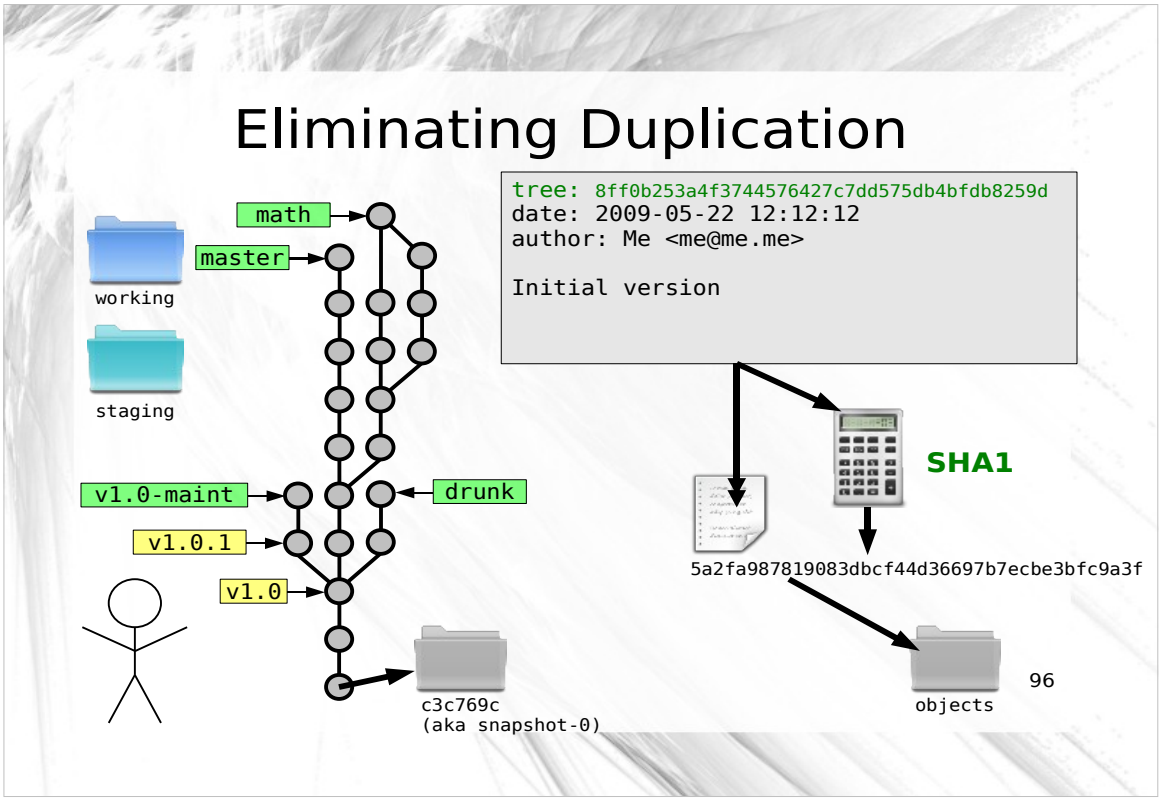

Once this has been accomplished for every directory and file in the snapshot, you have the SHA1 of the directory listing for the root folder of your snapshot.

Now you record the root tree's SHA1 somewhere, but where?



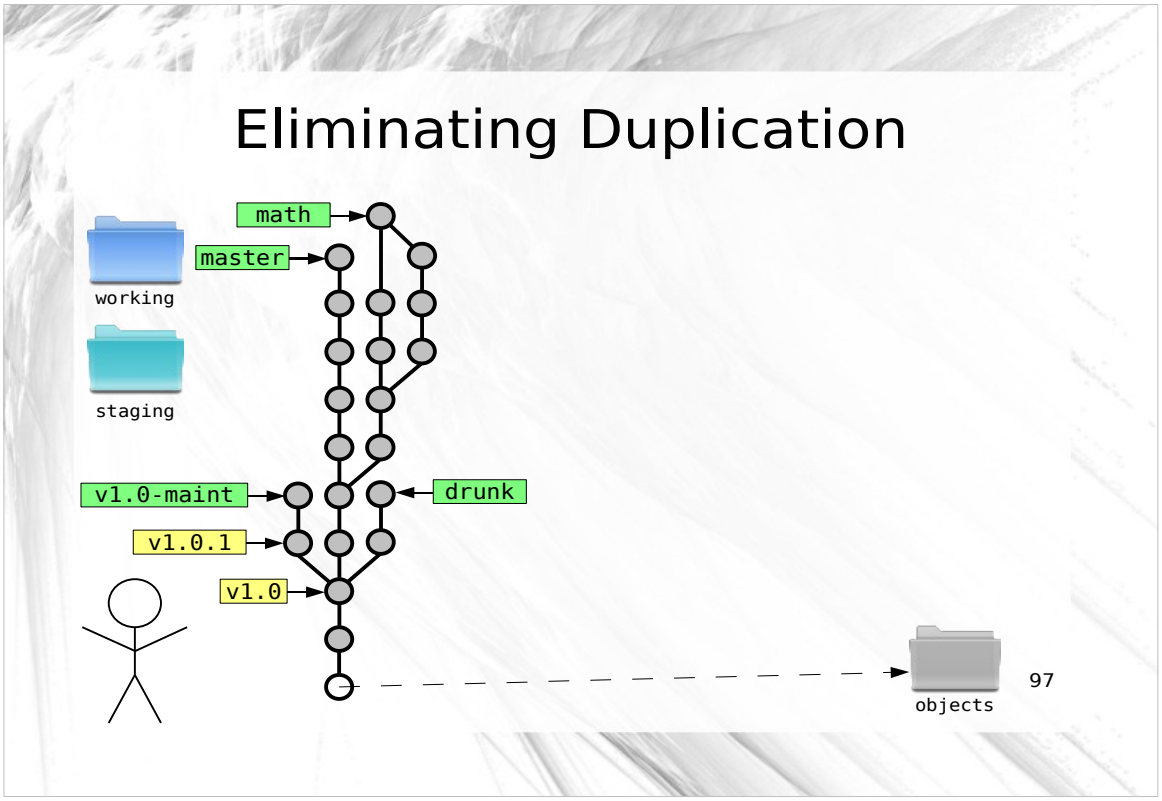

An ideal place to store it is in the snapshot message file.

This way, the uniqueness of the SHA1 of the message also depends on the entire contents of the snapshot, and you can guarantee with absolute certainty that two identical snapshot message SHA1s contain the same files!




It’s also convenient to create an object from the snapshot message in the same way that you do for blobs and trees.

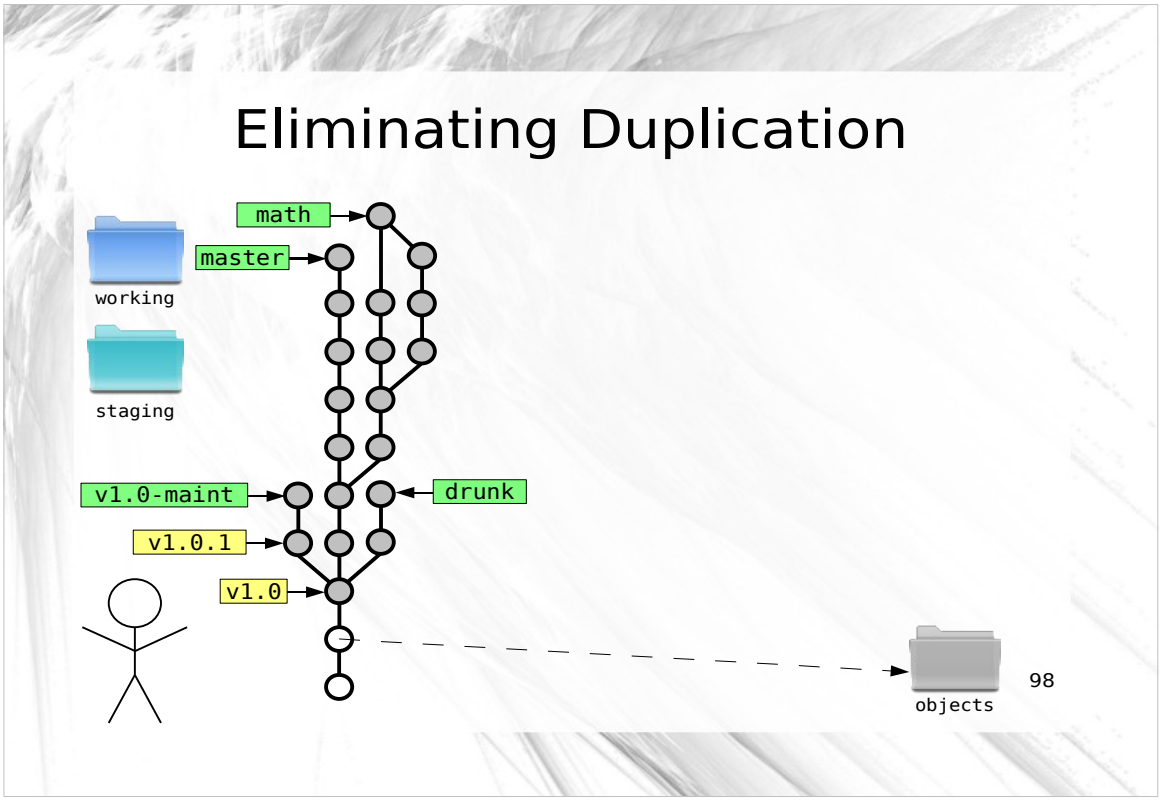
Since you’re maintaining a list of branch and tag names that point to message SHA1s you don’t have to worry about losing track of which snapshots are important to you




With all of this information stored in the objects directory, you can safely delete the snapshot directory that you used as the source of this operation.

If you want to reconstitute the snapshot at a later date it’s simply a matter of following the SHA1 of the root tree stored in the message file and extracting each tree and blob into their corresponding directory and file.

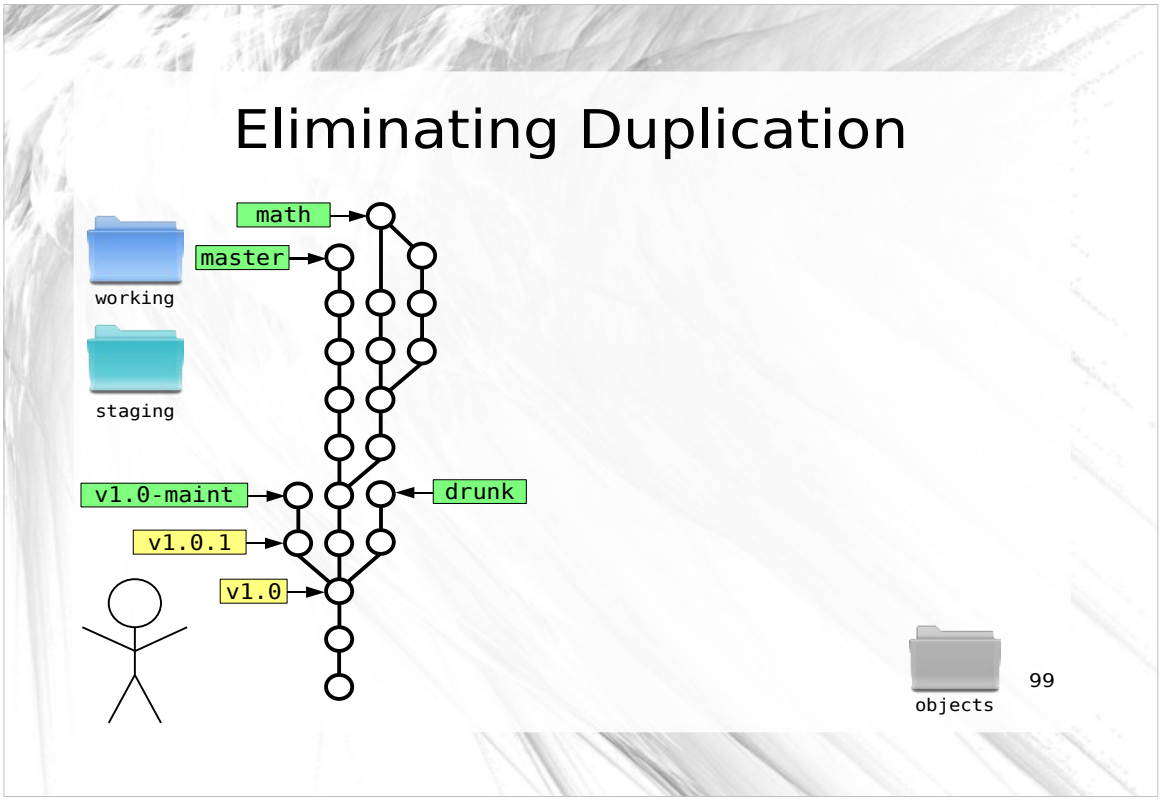
For a single snapshot, this transformation process doesn’t get you much. You’ve basically just converted one filesystem into another and created a lot of work in the process.




But once you start tranforming the second snapshot you discover something:

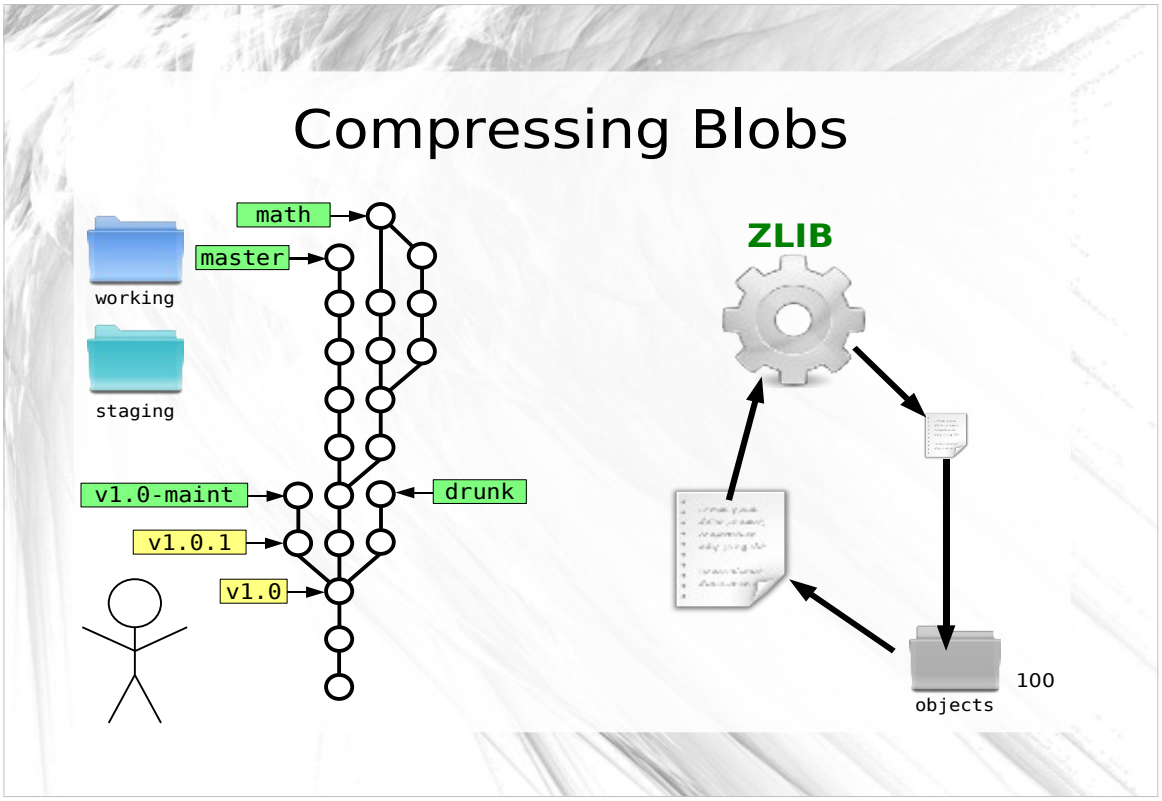
Many of the files and directories are unchanged from the first snapshot, and they therefore end up with the same SHA1. But since they have the same SHA1, they already exist in the objects directory, so you don't have to copy them in there.

Imagine two sequential snapshots in which only a single file in the root directory has changed. If the snapshots both contain 10 directories and 100 files, the transformation process will create 10 trees and 100 blobs from the first snapshot but only one new blob and one new tree from the second snapshot!



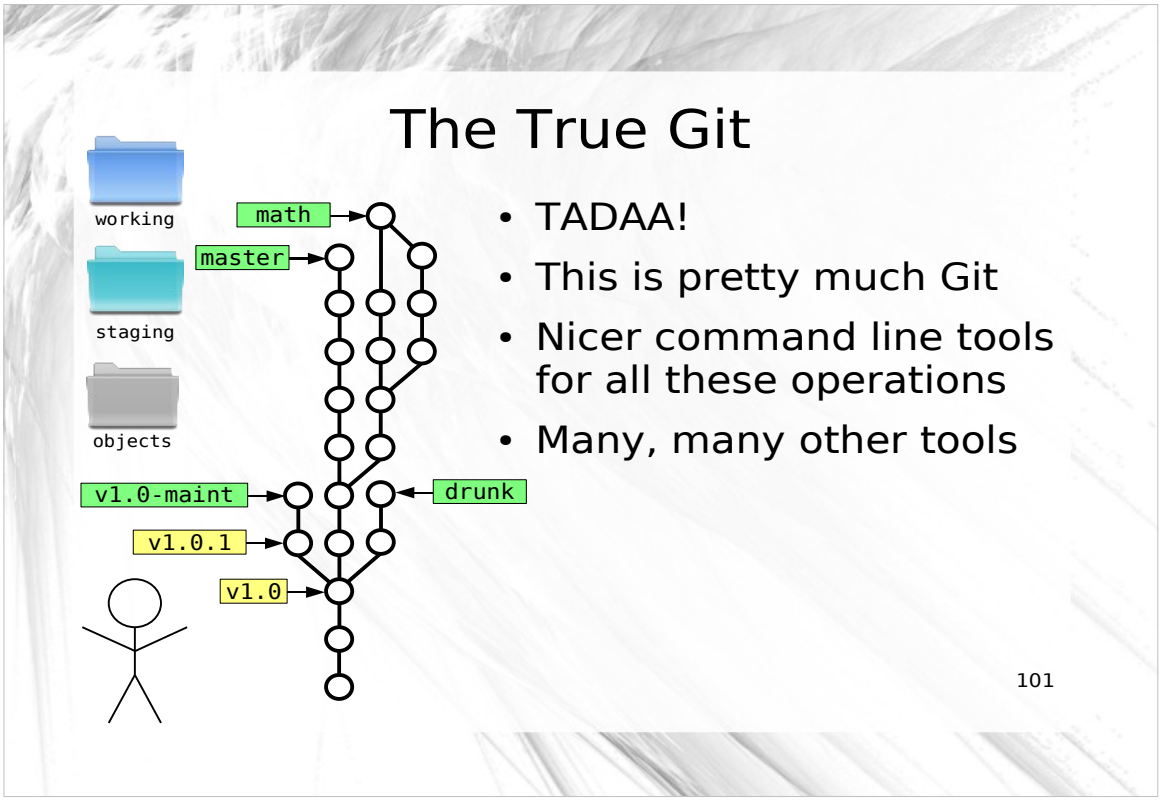

As you can see, the real benefits of this system arise from reuse of trees and blobs across snapshots.

By converting every snapshot directory in the old system to object files in the new system, you can drastically reduce the number of files that are stored on disk. Now, instead of storing perhaps 50 identical copies of a rarely changed file, you only need to keep one.




Eliminating blob and tree duplication significantly reduces the total storage size of your project history, but that’s not the only thing you can do to save space.

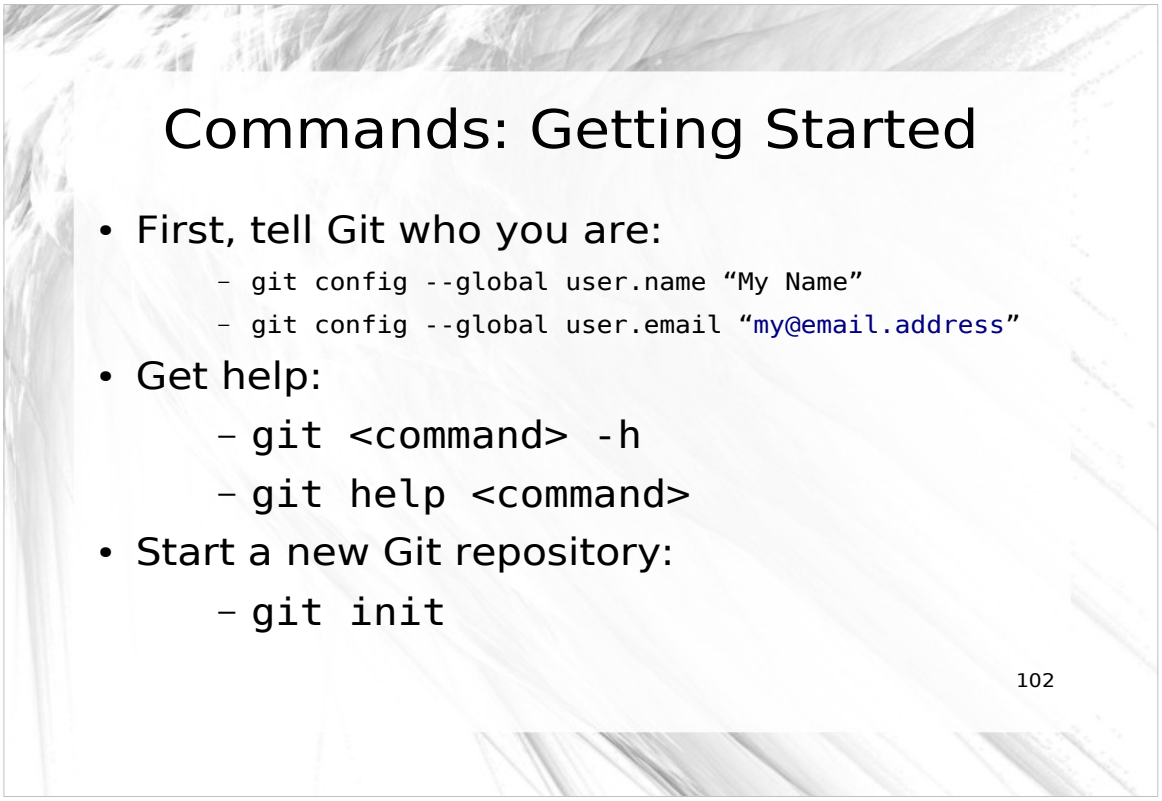
Source code is just text. Text can be very efficiently compressed using sommon compression algorithms. If you compress every blob before computing its SHA1 and saving it to disk you can further reduce the total storage size of the project history by a significant amount.




TADAA!

The VCS you have constructed is now a reasonable approximation of Git.

The main difference is that Git gives you very nice command lines tools to handle such things as creating new snapshots and switching to old ones (Git uses the term “commit” instead of “snapshot”), tracing history, keeping branch tips up-to-date, fetching changes from other people, merging and diffing branches, and hundreds of other common (and not-so-common tasks).

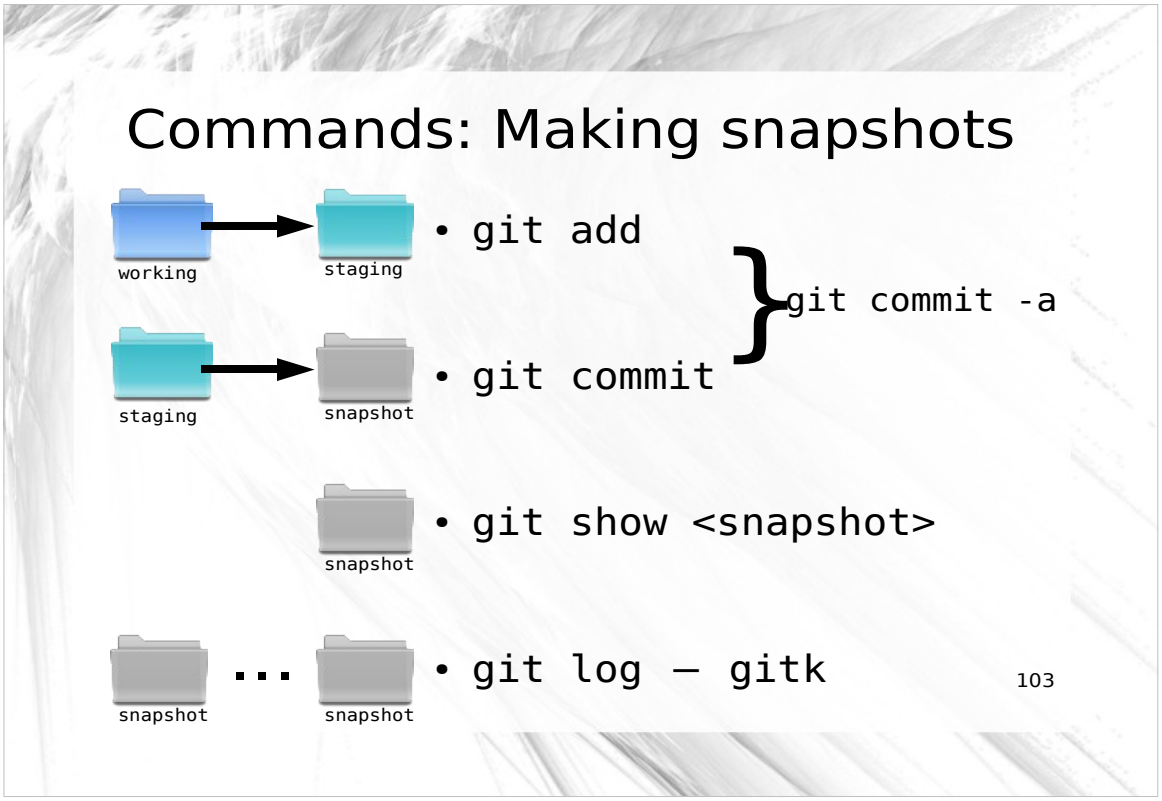



Finally, to get you started with simple usage of Git, here are some basic commands.

First, there are a couple of commands to make sure that your snapshots get the correct author information. You only need to run this once per machine you work on.

You can always get help on specific commands. “-h” gives you a short overview, “git help” gives you the reference docs for a command.

To start a new repo in the current directory, run “git init”. The current directory will be your working directory, and the objects, staging area, branches, tags, and other stuff will live in the “.git” subdirectory.

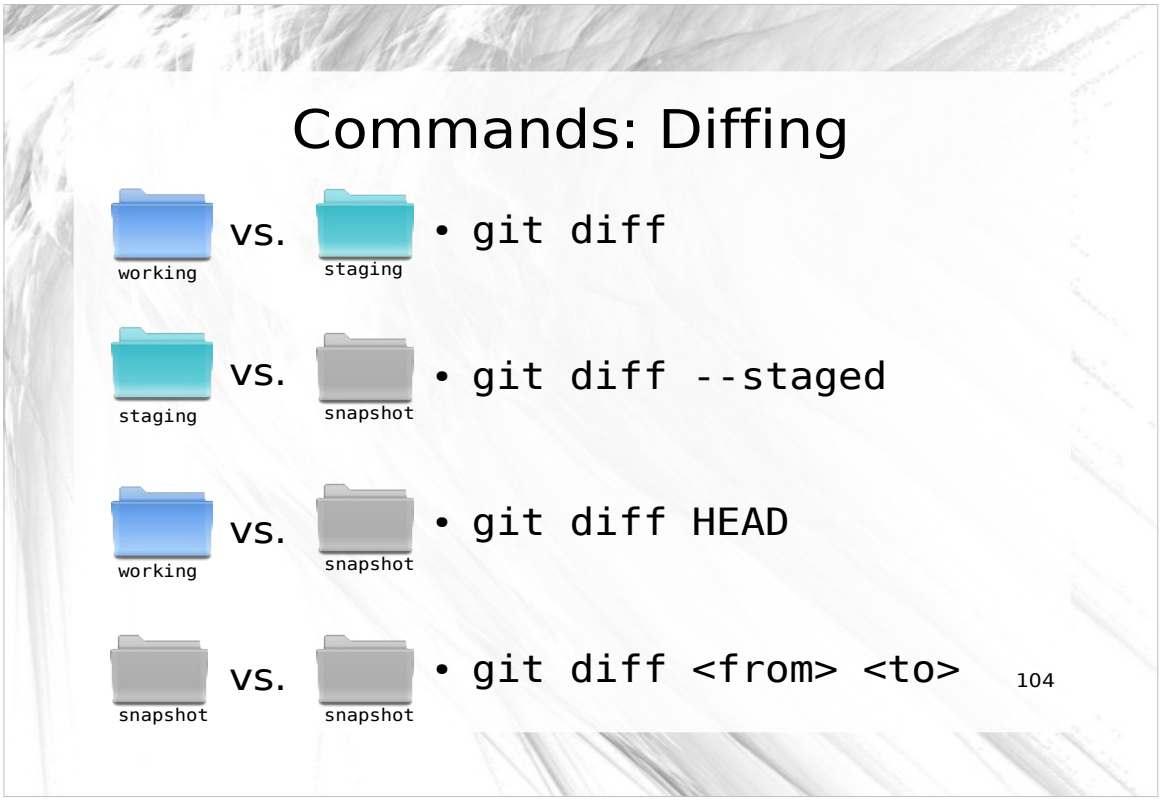



To move changes from your working directory into the staging area, use “git add”.

To make a new snapshot from the staging area, use “git commit”. Use “git commit -a” to make a new commit directly from all changed files in the working directory.

Use “git show” to show the contents of a given commit (the last commit on the current branch by default).

“git log” shows you a list of all commits on the current branch. “gitk” is a GUI version of this, that also draws a nice graph of the commits.

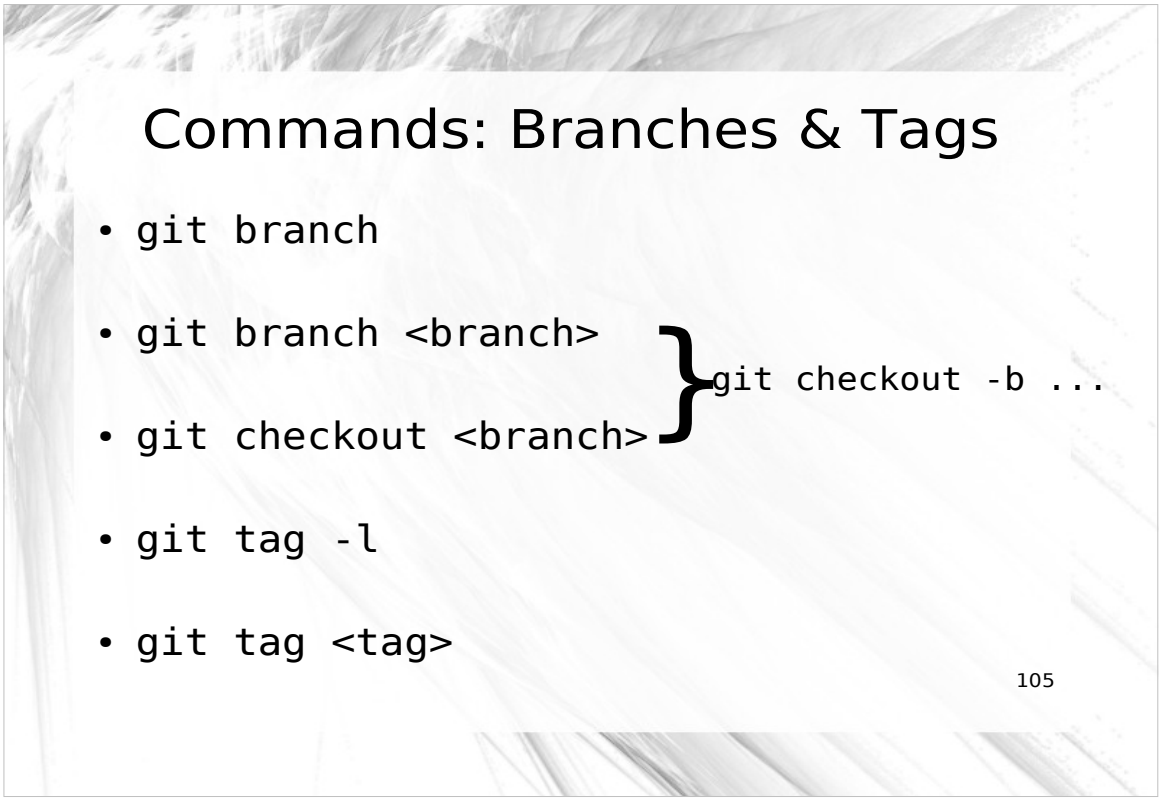



To see the differences between the working directory and the staging area, use “git diff”.

If you want to see the changes scheduled to become the next commit, use “git diff --staged”.

“git diff HEAD” shows you the diff between the working directory and the latest commit.

Finally you can use “git diff <from> <to>” to see the differences between any two commits in the repository.



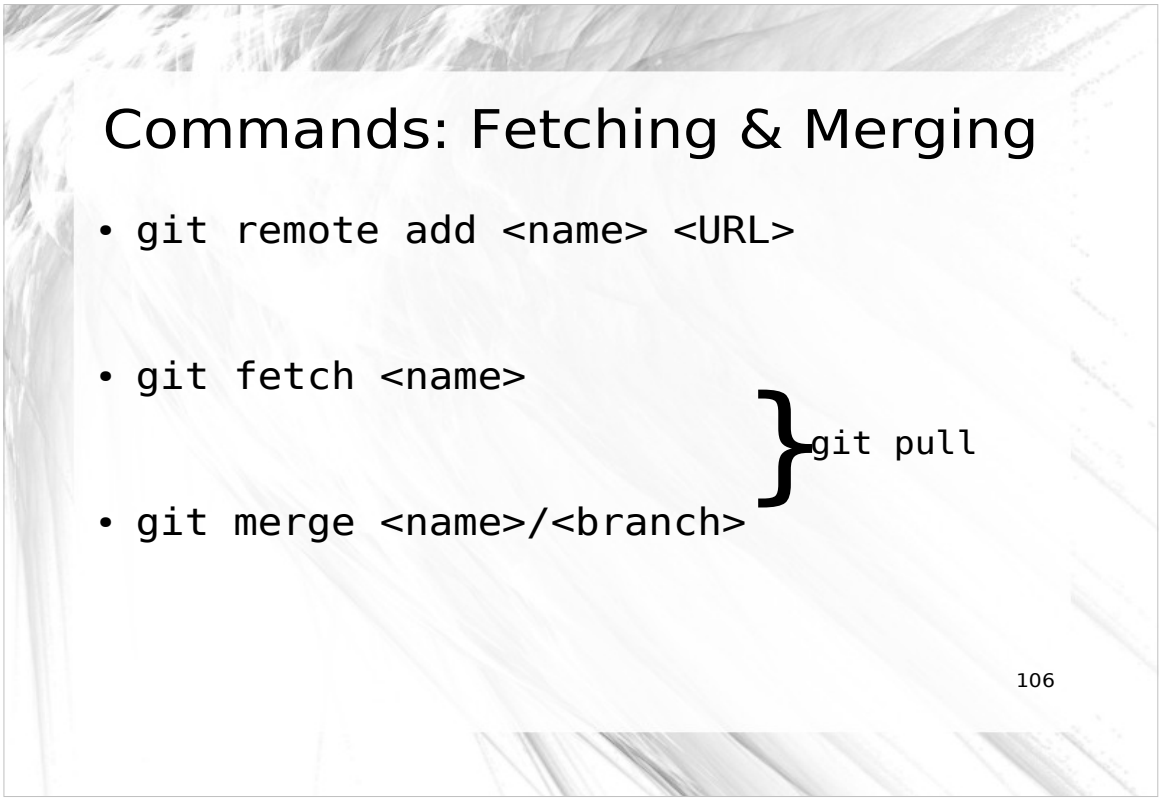

“git branch” shows a list of your current branches.

“git branch” is also used to create new branches.

Use “git checkout” to switch to a different branch. “git checkout -b” is a handy shortcut when you want to start working on a new branch.

“git tag -l” lists existing tags.

Use “git tag” to create new tags.

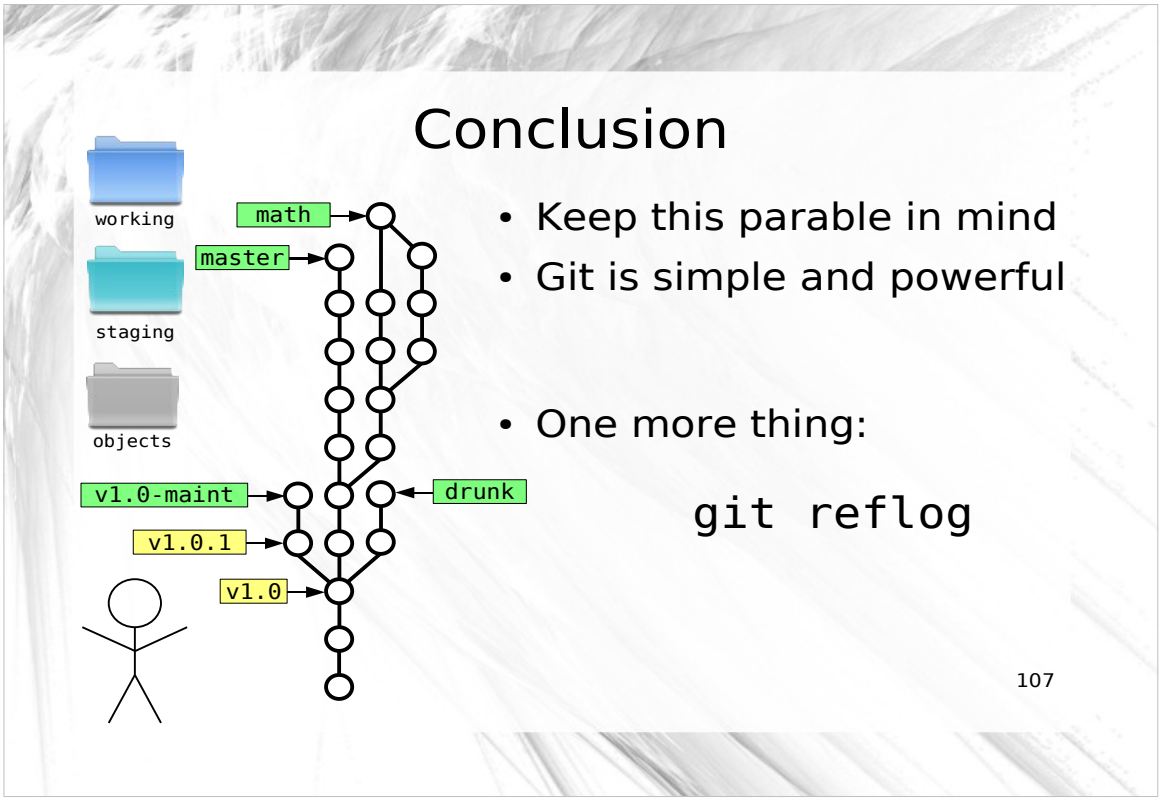



To start talking to another Git repo, use “git remote add”. This gives you a short name used to refer to the other repo (called a “remote”).

Use “git fetch” to fetch the branches in that repo. These are now available in your repo as “<name>/<branch>”.

To merge one of these branches into the current branch, use “git merge”.

git pull is a shorthand for those two commands, but don't use it if you don't want to merge.




As you continue to learn Git, keep this parable in mind. Git is really very simple underneath, and it is this simplicity that makes it so flexible and powerful.

One last thing before you run off to learn all the Git commands: remember that it is almost impossible to lose work that has been committed. Even when you delete a branch, all that's really happened is that the pointer to that commit has been removed. All of the snapshots are still in the objects directory, you just need to dig up the commit SHA1.

In these cases, look up "git reflog". It contains a history of what each branch pointed to and in times of crisis, it will save the day.

## Where to go next?

- @ Opera:
  - http://dvcs.oslo.osa
  - dvcs-interest@opera.com
  - johanh@opera.com
- In general:
  - Git homepage: http://git-scm.com
  - Git Community Book: http://book.git-scm.com/
  - http://gitorious.org or http://github.com



Here are some resources that you should follow as your next step.

Now, go, and become a Git master!

# Questions?

- Thanks for your attention!
- This presentation is available at:
  http://dvcs.oslo.osa/~johanh/git_parable/
- Reach me at <johanh@opera.com>