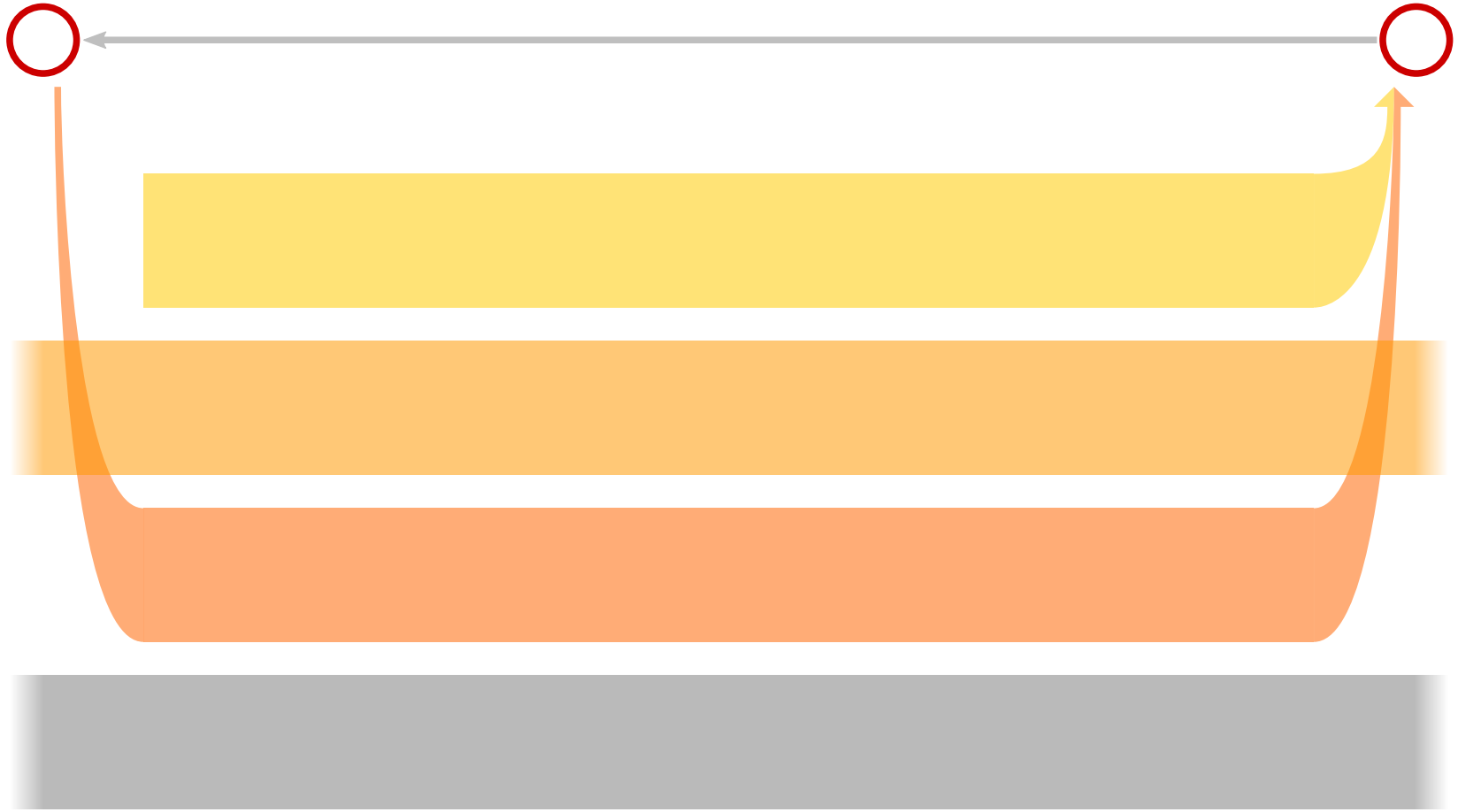
git

What happens at the file-level



git

What happens at the file-level

The 5 (well, 4) file states

Or is it 2?

The circle of life (of a file)

From one commit to another

Anatomy of a commit

With all the bloody details

Knowing what is happening (from the command line)

Can be usefull… sometimes

Undoing things

Can be usefull… too

Going the extra mile

With more complex yet super cool commands

git

What happens at the file-level

The 5 (well, 4) file states
Or is it 2?

The circle of life (of a file)
From one commit to another

Anatomy of a commit
With all the bloody details

Knowing what is happening (from the command line)
Can be usefull… sometimes

Undoing things
Can be usefull… too

Going the extra mile
With more complex yet super cool commands

In git, just like in every journey, you must start somewhere…

```
mkdir somewhere && cd somewhere && ls -a
.   ..
```

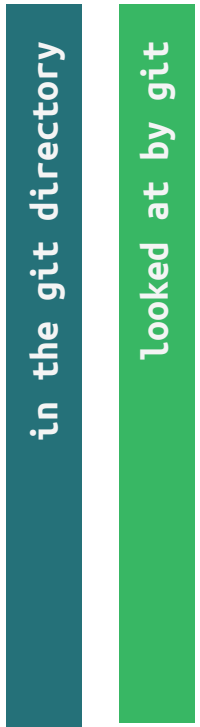
… and do the first step

```
git init && ls -a
.   ..   .git
```

All files in the directory are in one of the 5 different states

First, the files git will not care about

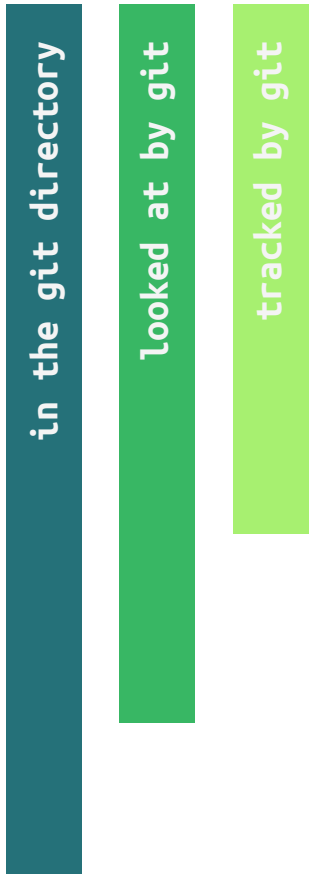
These files are notably defined in the ./.gitignore file



ignored git won't look at these files

Then, files git looks at, but does not "track"

Tracking means recording the successive versions, i.e. taking snapshots of



untracked git doesn't record changes in these files,
but knows whether they exist – it's the default state for new files

ignored git won't look at these files

Then files tracked by git, that come in 3 states

One should note that files *are not automatically included* in the next snapshot

Recap: File states

For git, **a file is either tracked** (i.e. git records its successive versions), **or not.**

Tracked files are in one of these states:

- *staged*
- *modified*
- *unmodified*

Non-tracked files are either:

- *untracked*
- *ignored*

git

What happens at the file-level

- The 5 (well, 4) file states
  - Or is it 2?
- **The circle of life (of a file)**
  - **From one commit to another**
- Anatomy of a commit
  - With all the bloody details
- Knowing what is happening (from the command line)
  - Can be usefull… sometimes
- Undoing things
  - Can be usefull… too
- Going the extra mile
  - With more complex yet super cool commands

Let's take the journey of a file

staged

modified

unmodified

untracked

New files are are created in the "untracked" state

staged

modified

unmodified

untracked

git doesn't record changes in these files,
but knows whether they exist — it's the default state for new files

```
touch hi.txt && ls
hi.txt
```

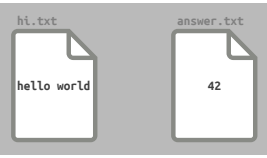
Git doesn't care about untracked files modification

staged

modified

unmodified

untracked




```
echo "hello world" > hi.txt
echo "42" > answer.txt
```

A file must be "added" to a snapshot to be tracked by git

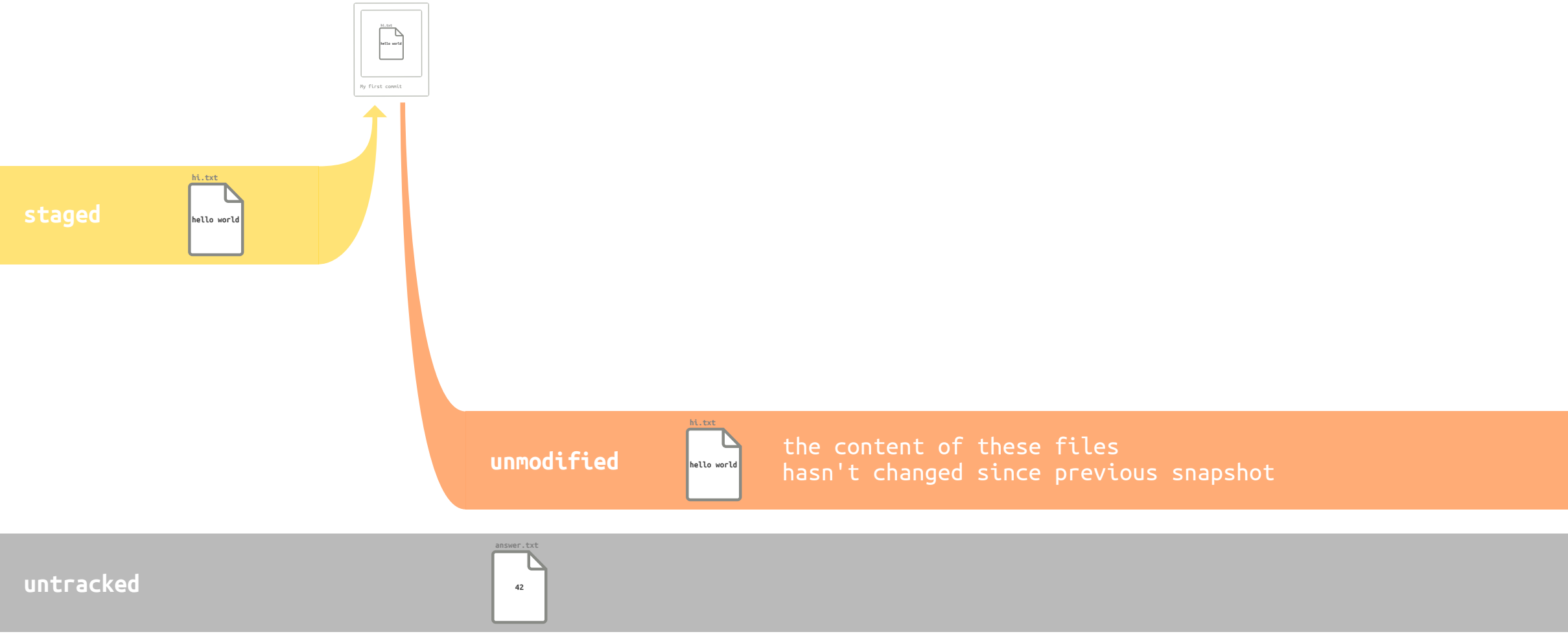

```
git add hi.txt
```

Taking a snapshot it done by creating a "commit"

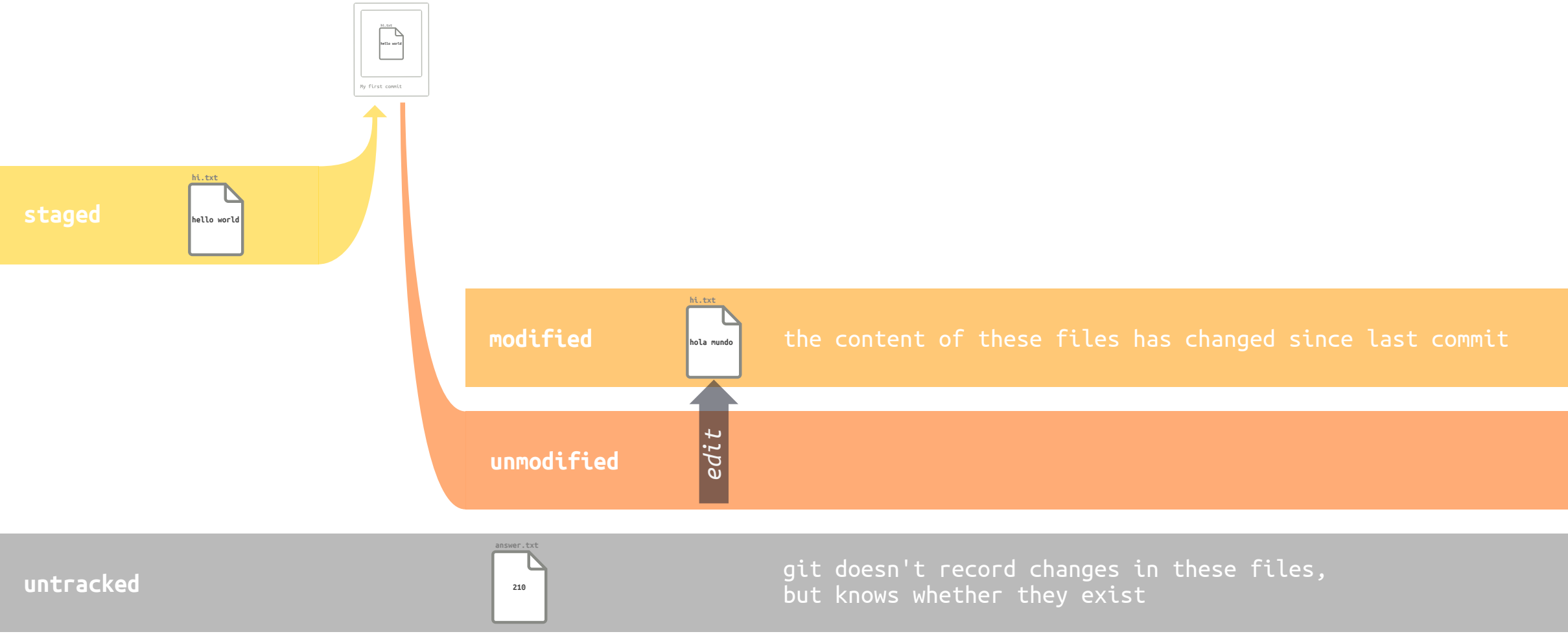
Staged files are included in the commit, but not untracked ones


```
git commit -m "My first commit"
```

After the commit, a staged file turns into "unmodified"…



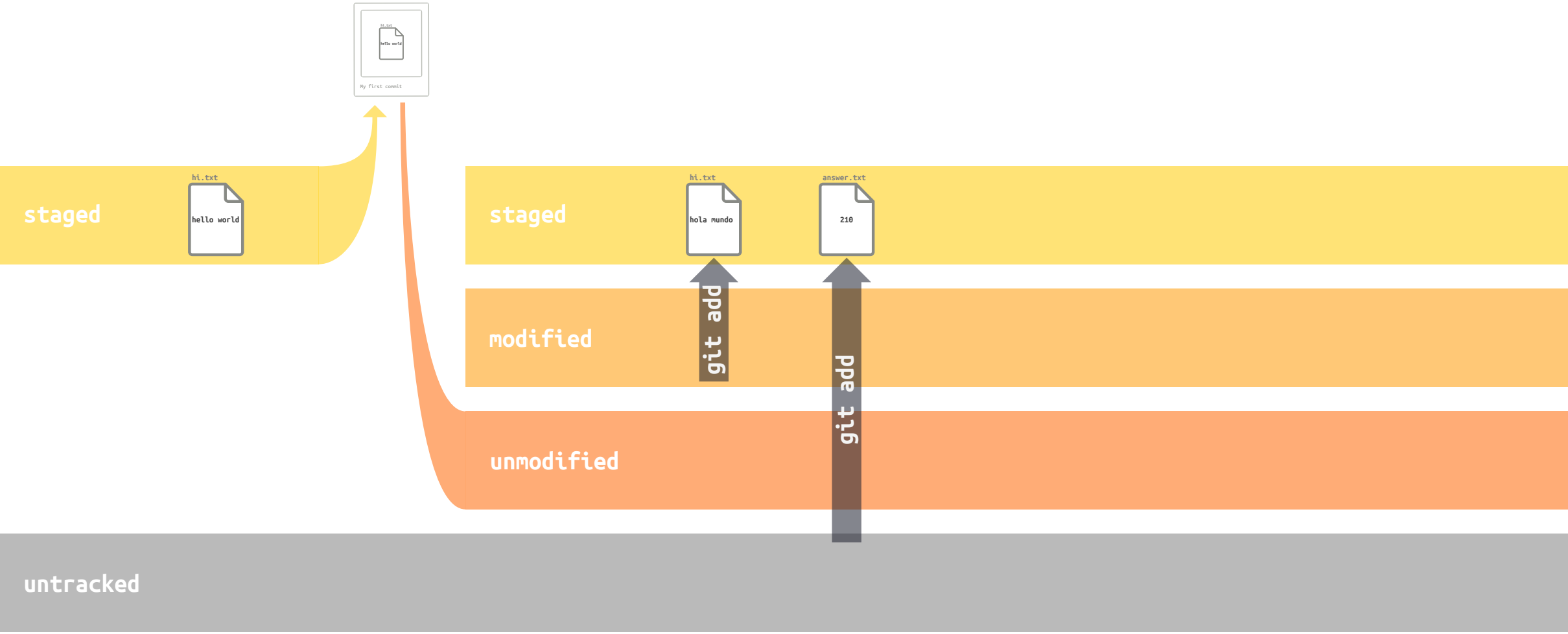
… until it is modified




```
echo "hola mundo" > hi.txt
echo "210" > answer.txt
```

… and then staged again.

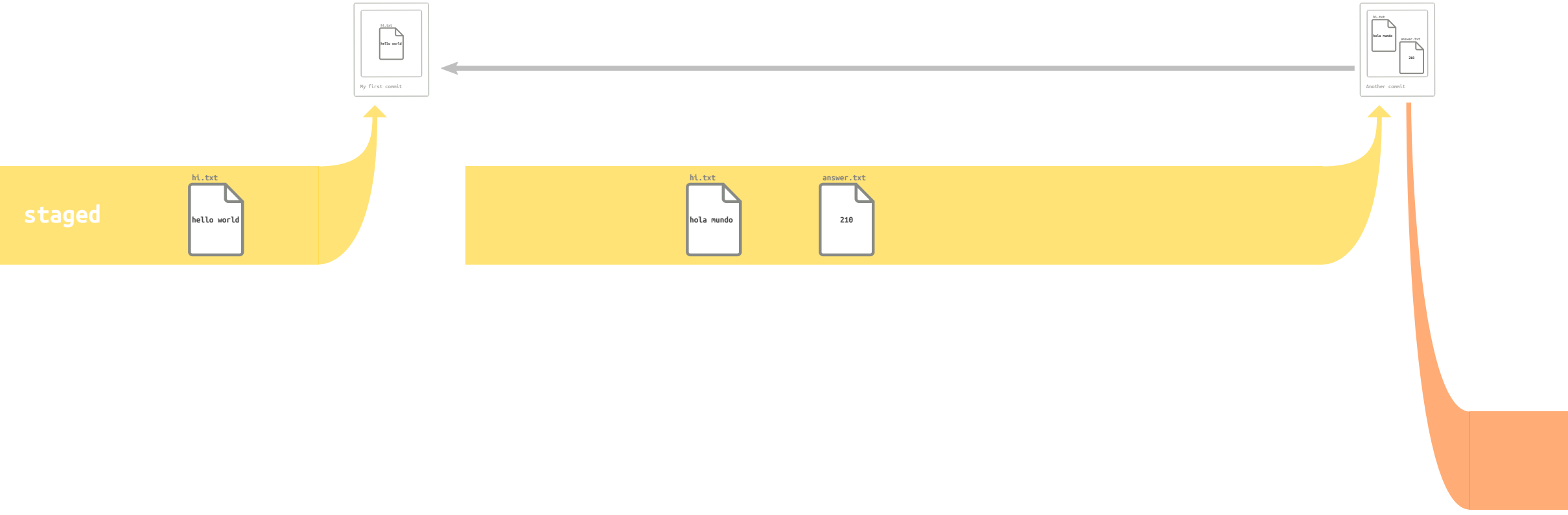
Note how we use the same command to stage from "untracked" and "modified" states




```
git add hi.txt
git add answer.txt
```

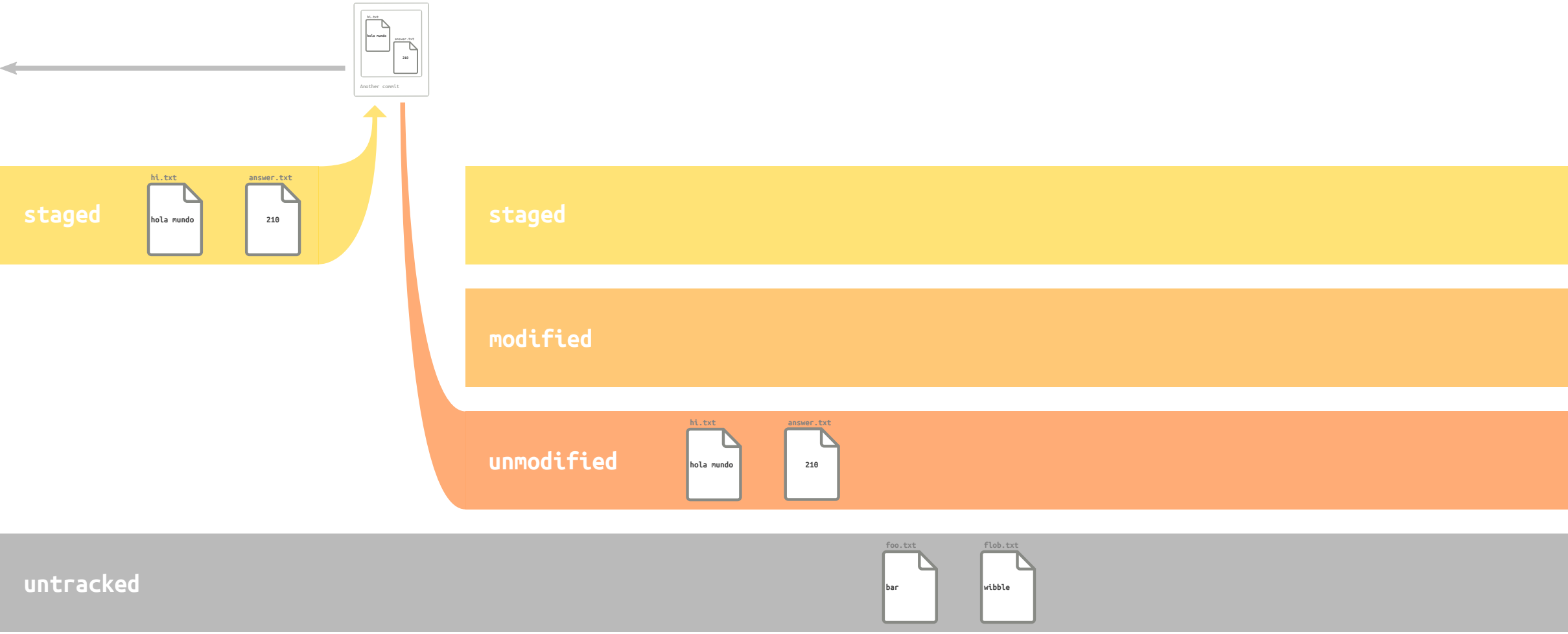
Lets take another snapshot

Note how both snapshots are linked with an arrow meaning *"is the child of"*



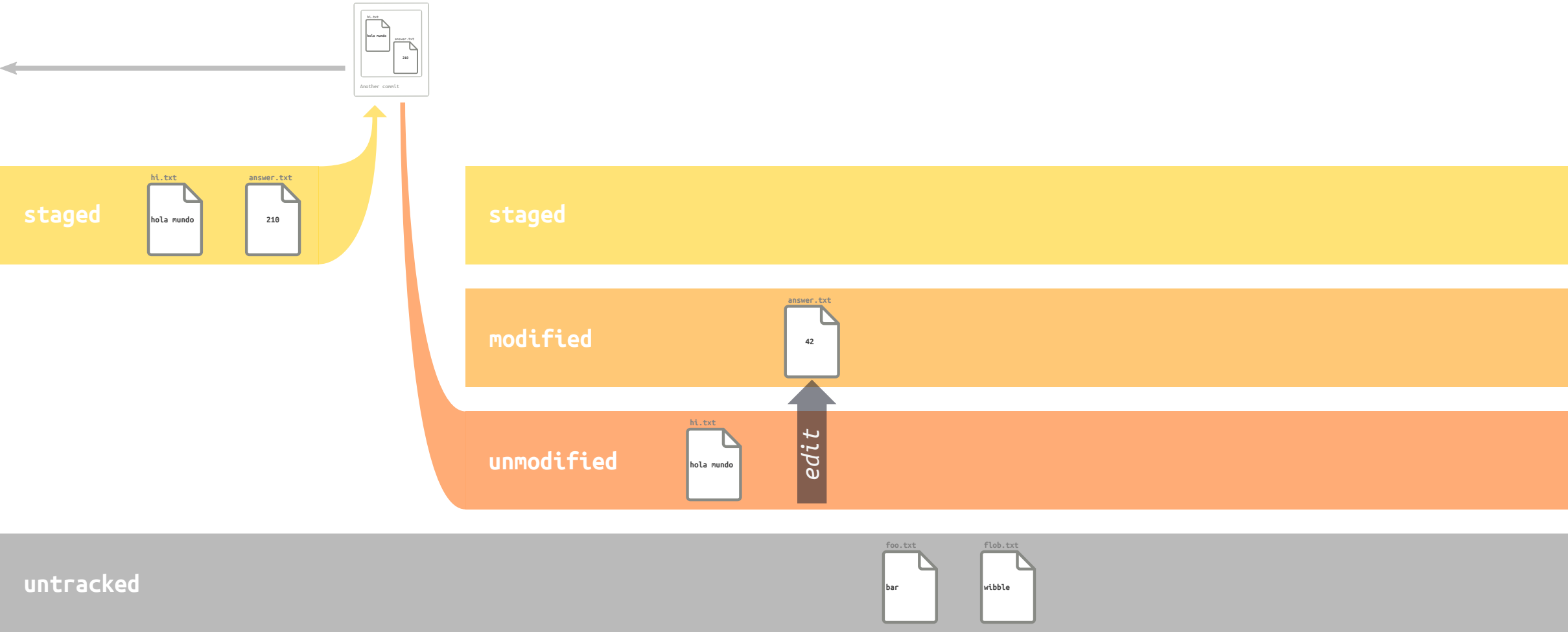

```
git commit -m "Another commit"
```

Life goes on…



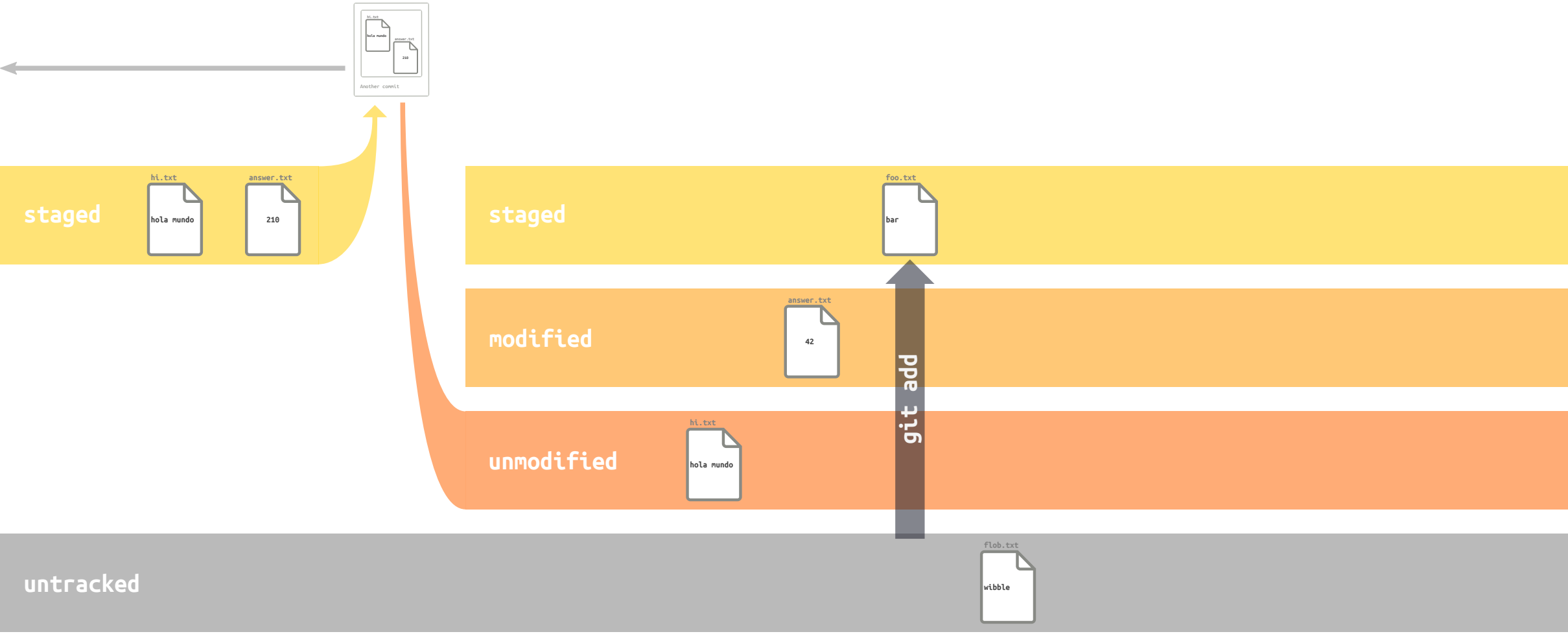

```
echo "bar" > foo.txt
echo "wibble" > flob.txt
```

Life goes on…




```
echo "42" > answer.txt
```

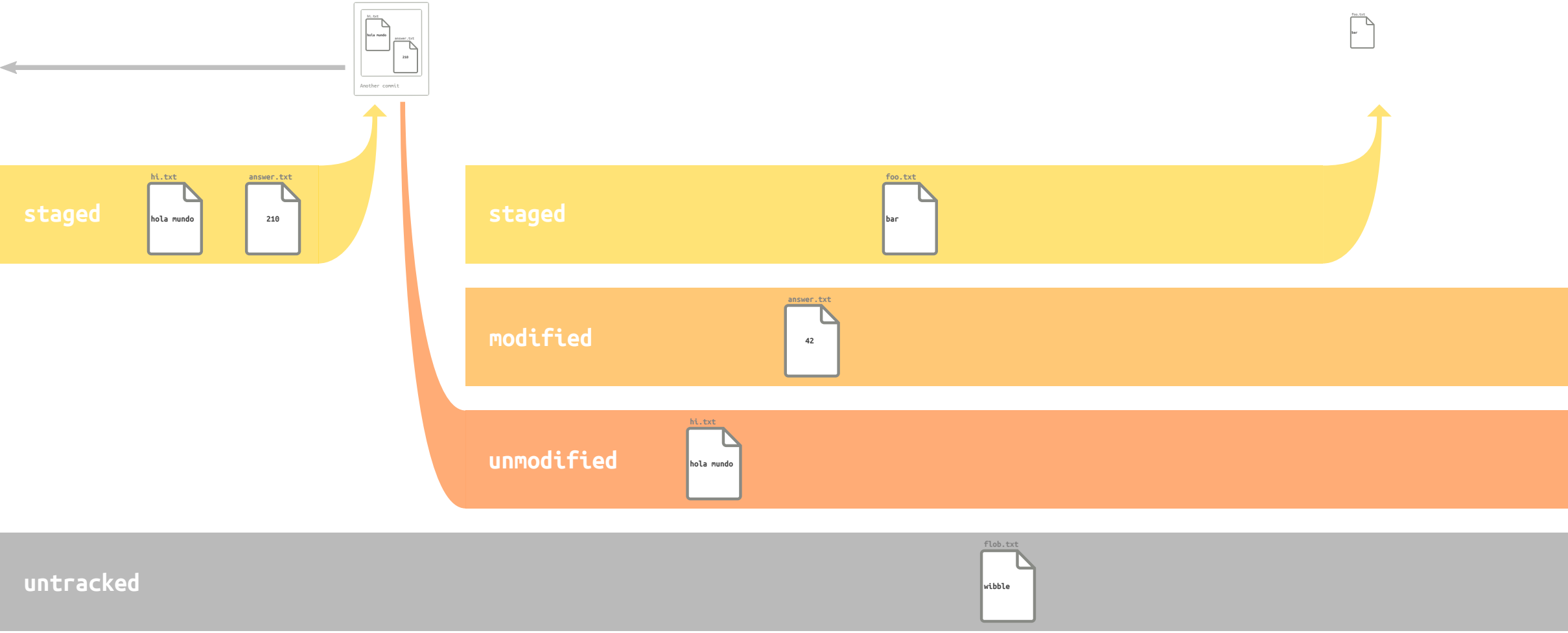
Life goes on…




```
git add foo.txt
```

Creating a commit with files in multiple states

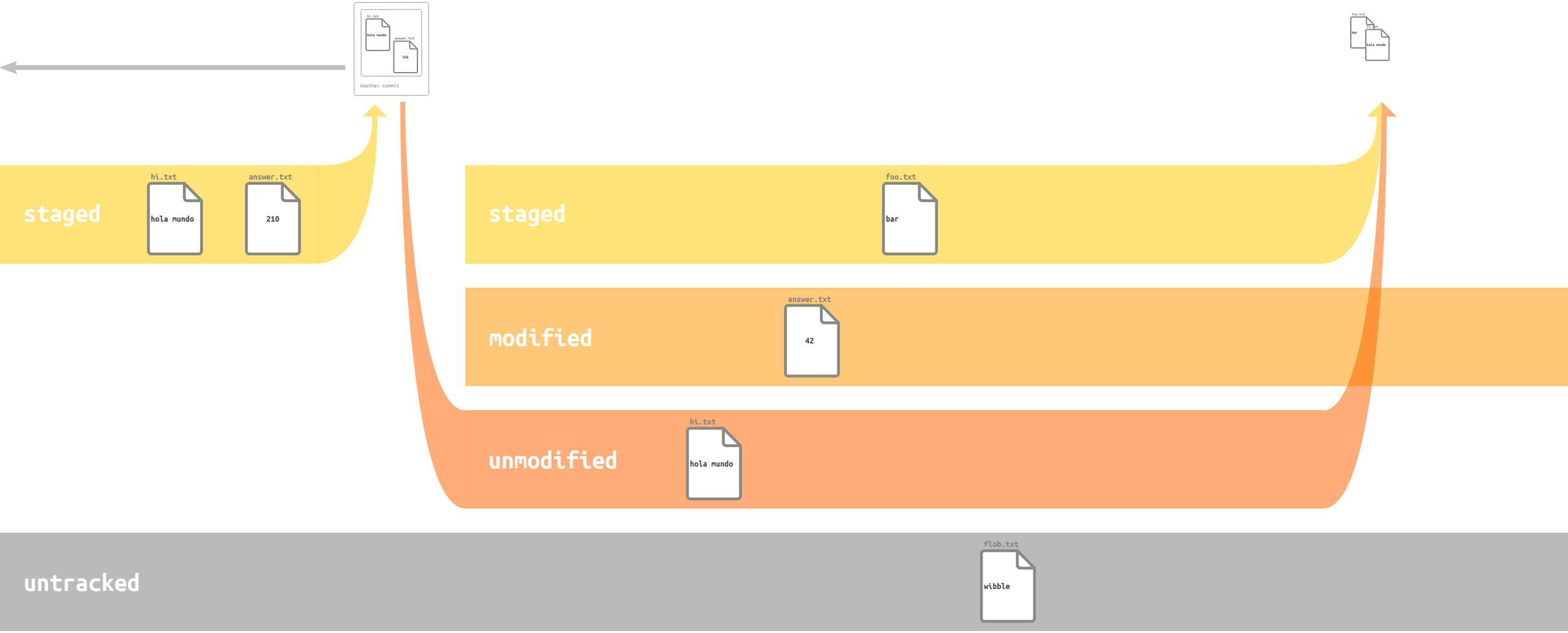
Staged files are obviously included…




```
git commit -m "This is more complicated"
```

Creating a commit with files in multiple states

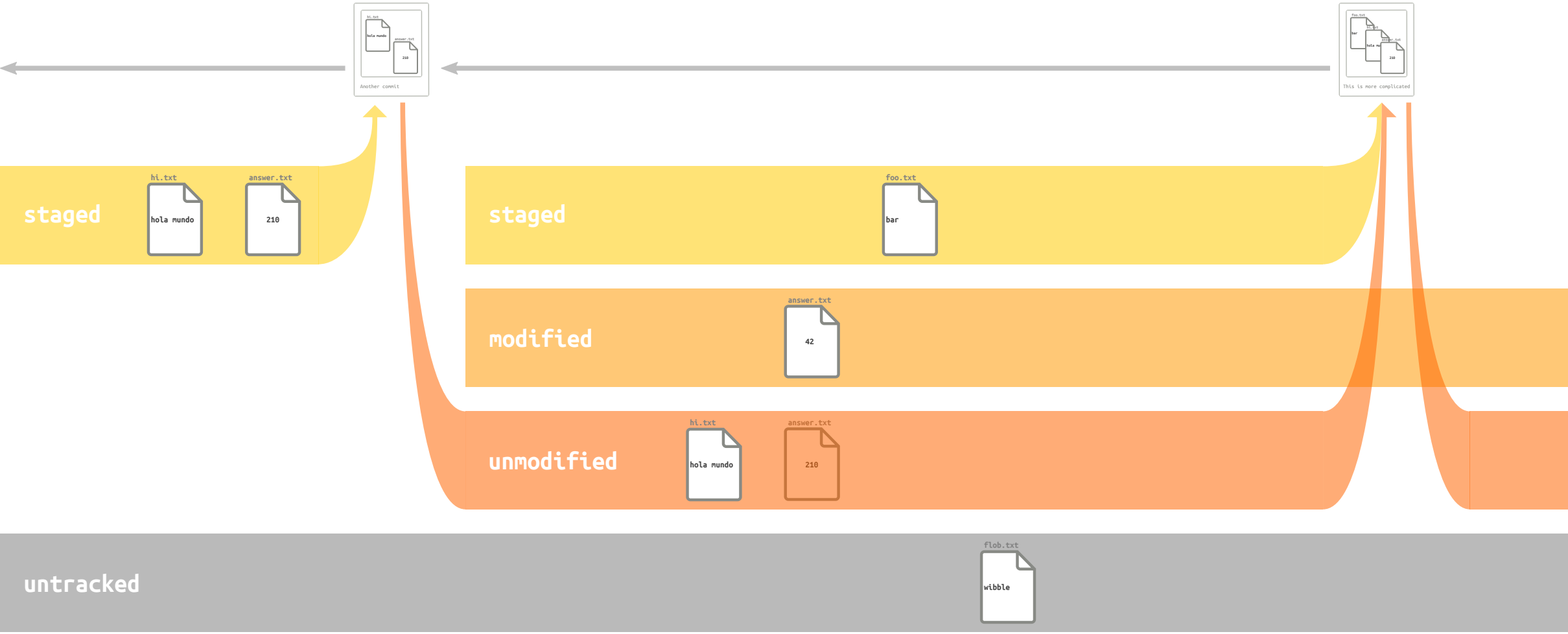
… unmodified files as well (as they weren't removed) …




```
git commit -m "This is more complicated"
```

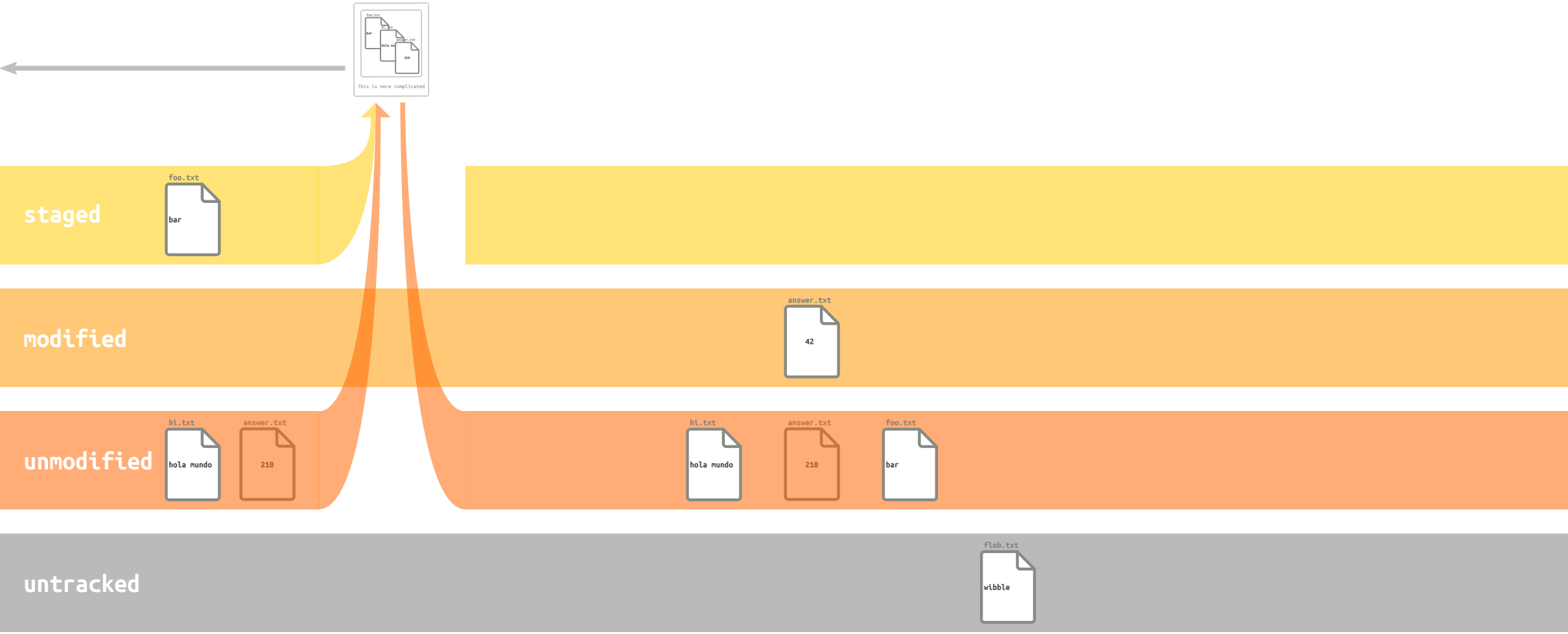
Creating a commit with files in multiple states

… along with the previous (unmodified) version of modified files.

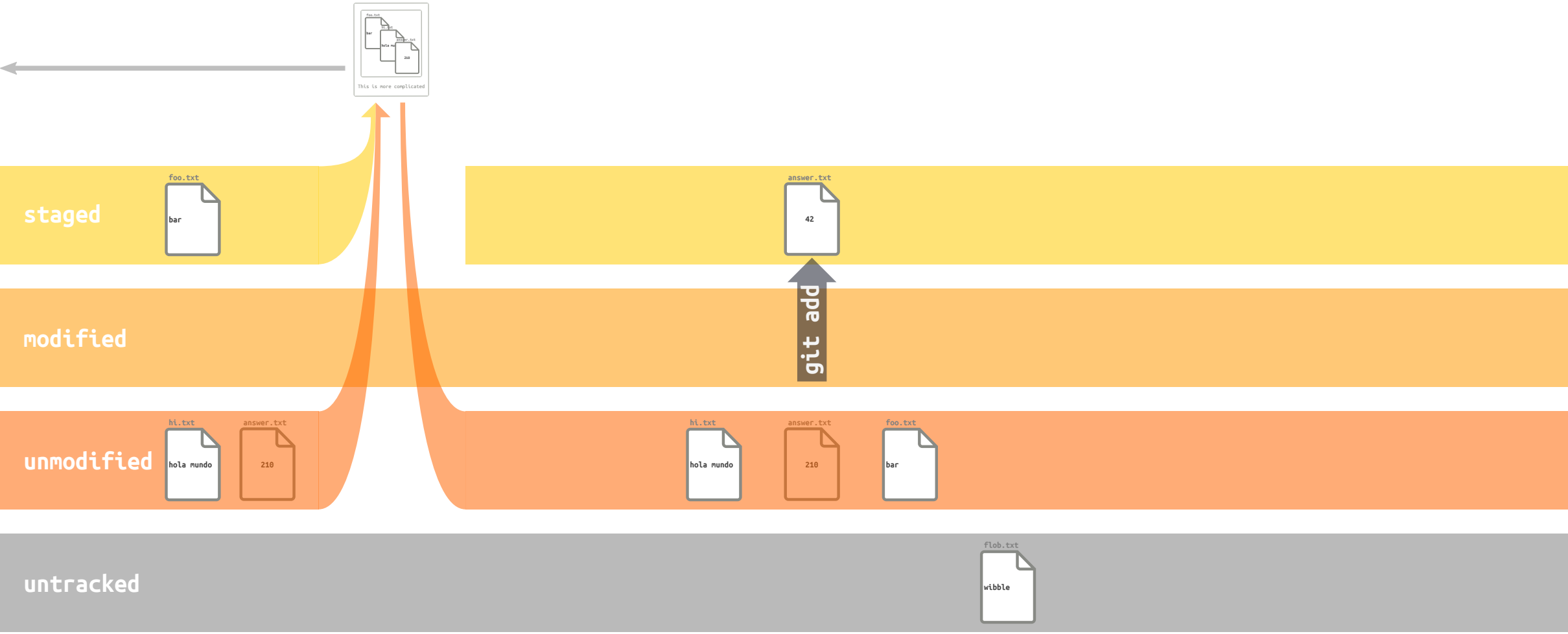



```
git commit -m "This is more complicated"
```

A "modified" file remains "modified" after a commit

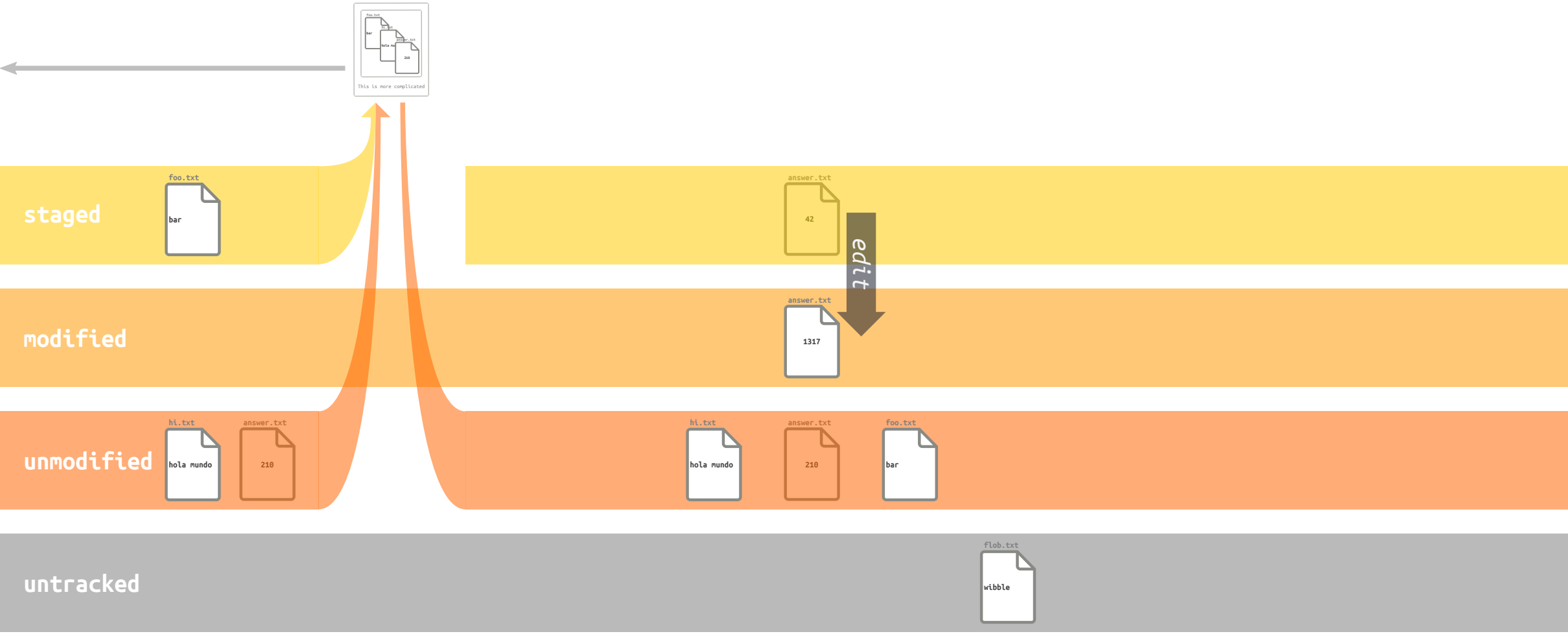


Something similar happens with staged-then-modified files



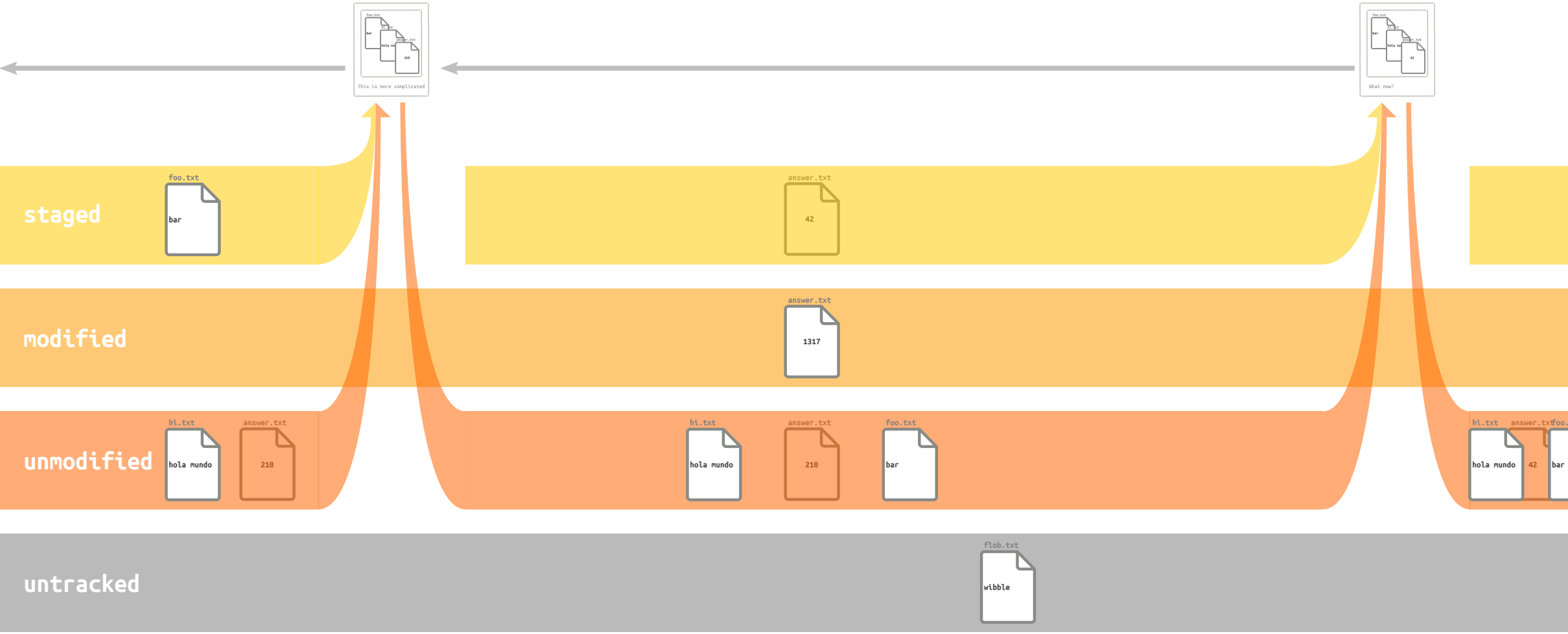

```
git add answer.txt
```

Something similar happens with staged-then-modified files




```
echo "1317" > answer.txt
```

Capturing file's version of a file actually happens at "git add"

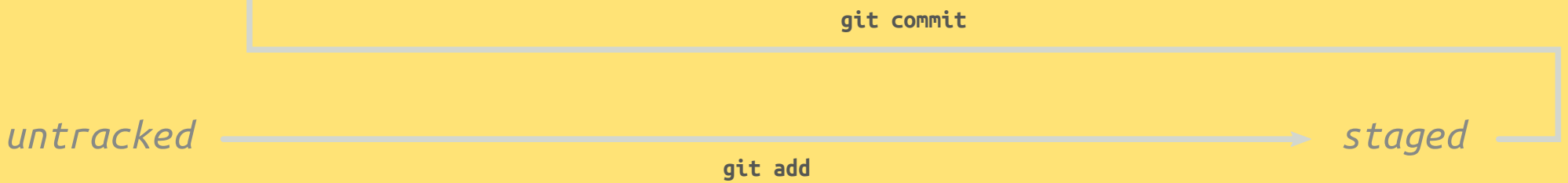



```
git commit -m "What now?"
```

Recap: Life-cycle of a file, during a commit

The typical life-cycle of a **tracked file** is:

The typical life-cycle of an **untracked file** is:

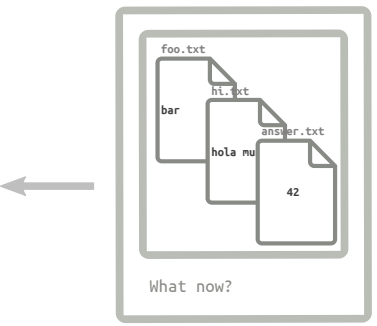


git

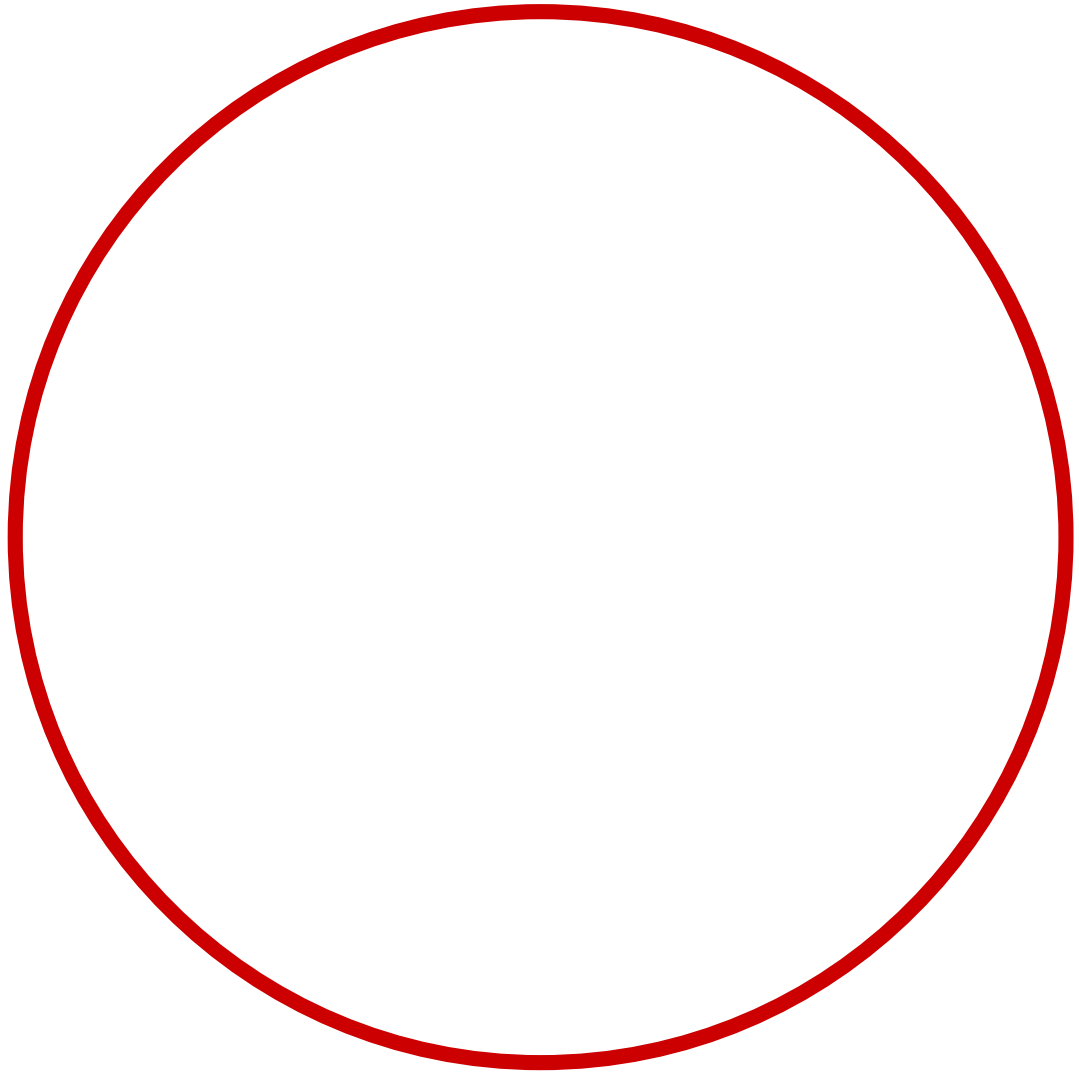
What happens at the file-level

- The 5 (well, 4) file states
  - Or is it 2?
- The circle of life (of a file)
  - From one commit to another
- **Anatomy of a commit**
  - **With all the bloody details**
- Knowing what is happening (from the command line)
  - Can be usefull… sometimes
- Undoing things
  - Can be usefull… too
- Going the extra mile
  - With more complex yet super cool commands

Anatomy of a commit

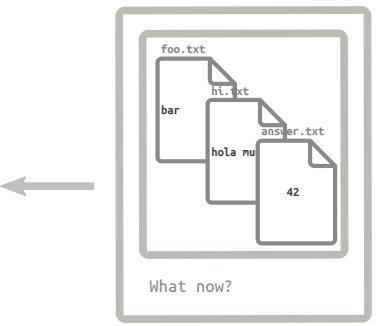


=



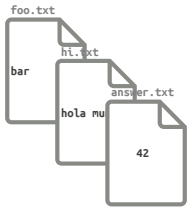
A commit is the snapshot

… but is also much more

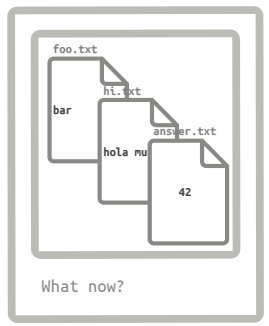


=

files snapshot

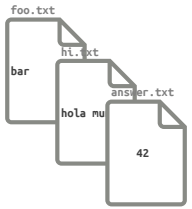


A commit is more than the snapshot



=

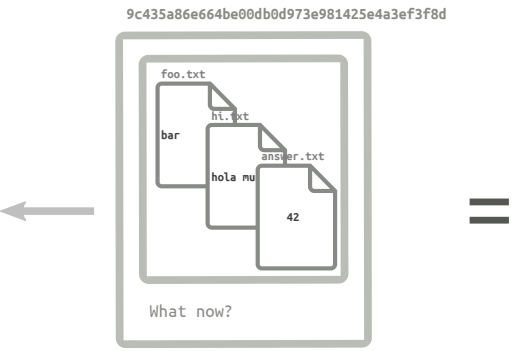
files snapshot



message What now?

A commit is more than the snapshot

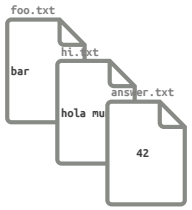
ID is a 40-character long hash (we'll come to that)



=

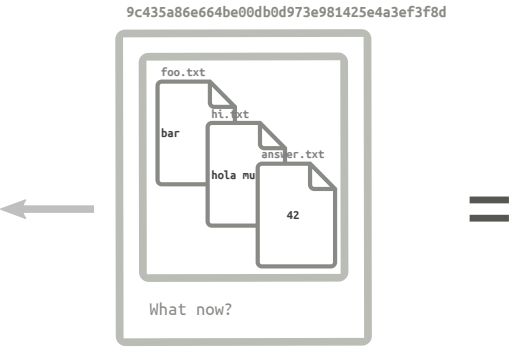
unique ID 9c435a86e664be00d

files snapshot



message What now?

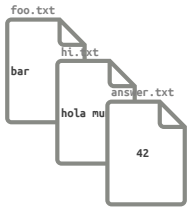
A commit is more than the snapshot



=

unique ID 9c435a86e664be00d

files snapshot



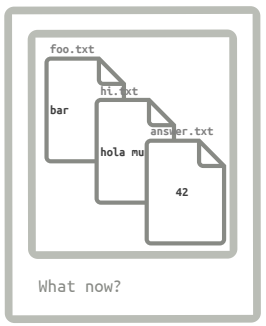
message What now?

ID of the parent(s) 0d973e9c4353ef3f8ddb98a8

A commit is more than the snapshot

Author's details are defined in ~/.gitconfig

9c435a86e664be00db0d973e981425e4a3ef3f8d

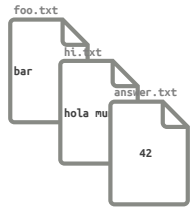


Étienne Boisseau-Sierra <ebosi@unipart.io>
Wed Oct 16 15:45:42 2019 +0100

=

unique ID 9c435a86e664be00d

files snapshot



message What now?

ID of the parent(s) 0d973e9c4353ef3f8ddb98a8

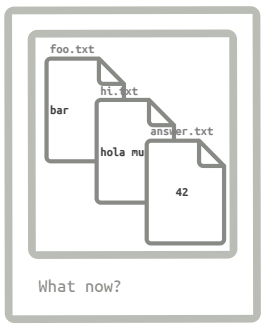
author Étienne Boisseau-Sierra <ebosi@unipart.io>
user.name user.email

date Wed Oct 16 15:45:42 2019 +0100

A commit is more than the snapshot

Committer's data usually are the same as Author's ones (except when, e.g., rebasing)

9c435a86e664be00db0d973e981425e4a3ef3f8d

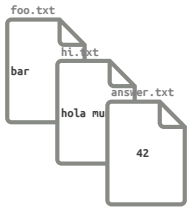


Étienne Boisseau-Sierra <ebosi@unipart.io>
Wed Oct 16 15:45:42 2019 +0100
Étienne Boisseau-Sierra <ebosi@unipart.io>
Wed Oct 16 15:45:42 2019 +0100

=

unique ID 9c435a86e664be00d

files snapshot



message What now?

ID of the parent(s) 0d973e9c4353ef3f8ddb98a8

author Étienne Boisseau-Sierra <ebosi@unipart.io>

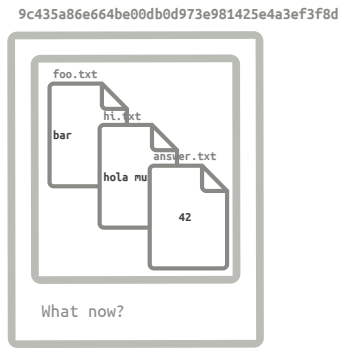
user.name user.email

date Wed Oct 16 15:45:42 2019 +0100

committer Étienne Boisseau-Sierra <e

committer date Wed Oct

A commit is more than the snapshot



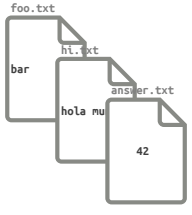
9c435a86e664be00db0d973e981425e4a3ef3f8d

Étienne Boisseau-Sierra <ebosi@unipart.io>
Wed Oct 16 15:45:42 2019 +0100
Étienne Boisseau-Sierra <ebosi@unipart.io>
Wed Oct 16 15:45:42 2019 +0100

=

unique ID 9c435a86e664be00d

files snapshot



message What now?

ID of the parent(s) 0d973e9c4353ef3f8ddb98a8

author Étienne Boisseau-Sierra <ebosi@unipart.io>
user.name user.email

date Wed Oct 16 15:45:42 2019 +0100

committer Étienne Boisseau-Sierra <e

committer date Wed Oct

unique ID defined as the checksum (i.e. SHA-1 hash) of all other metadata

this is key aspect of git's integrity

Recap: Anatomy of a git commit

A git commit is constituted of **files snapshot + metadata**.

If a **single element** (file content, date, author…) **changes**, the **commit ID will change**.

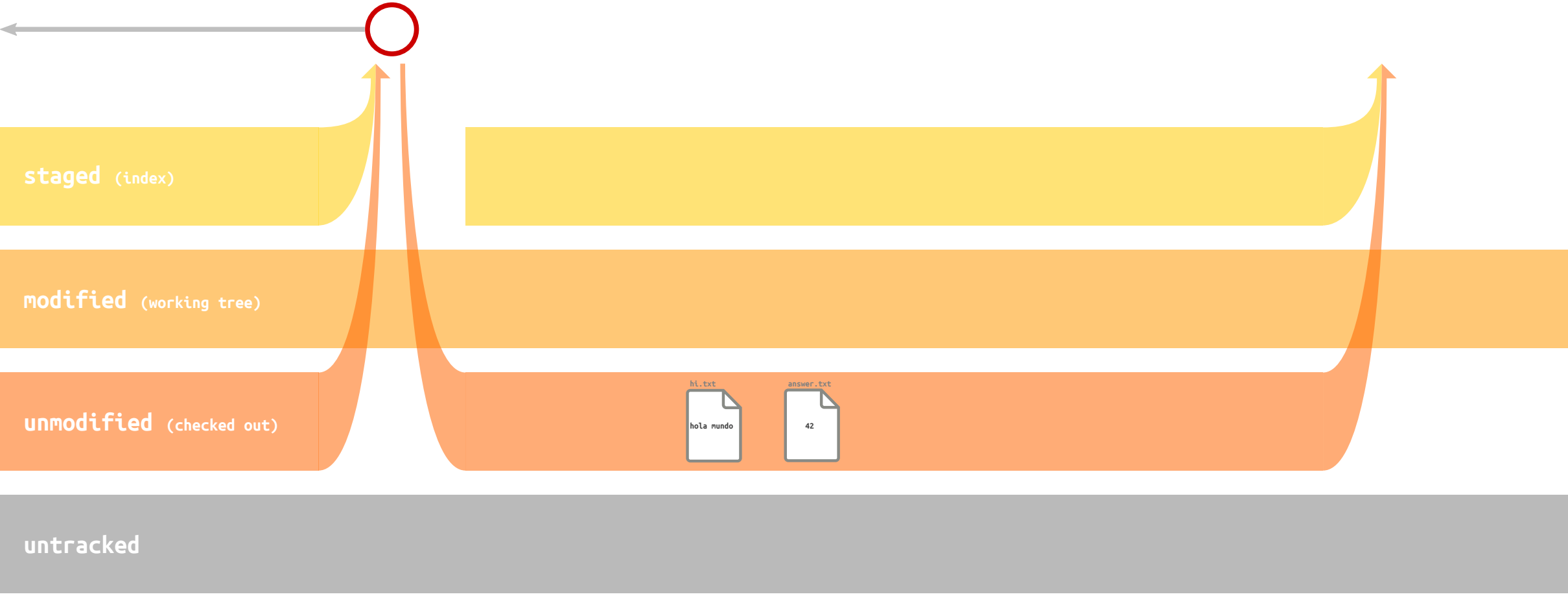
If two commits have **different ID**, **they are totally different** for git (even if they might have the same files snapshot).

git

What happens at the file-level

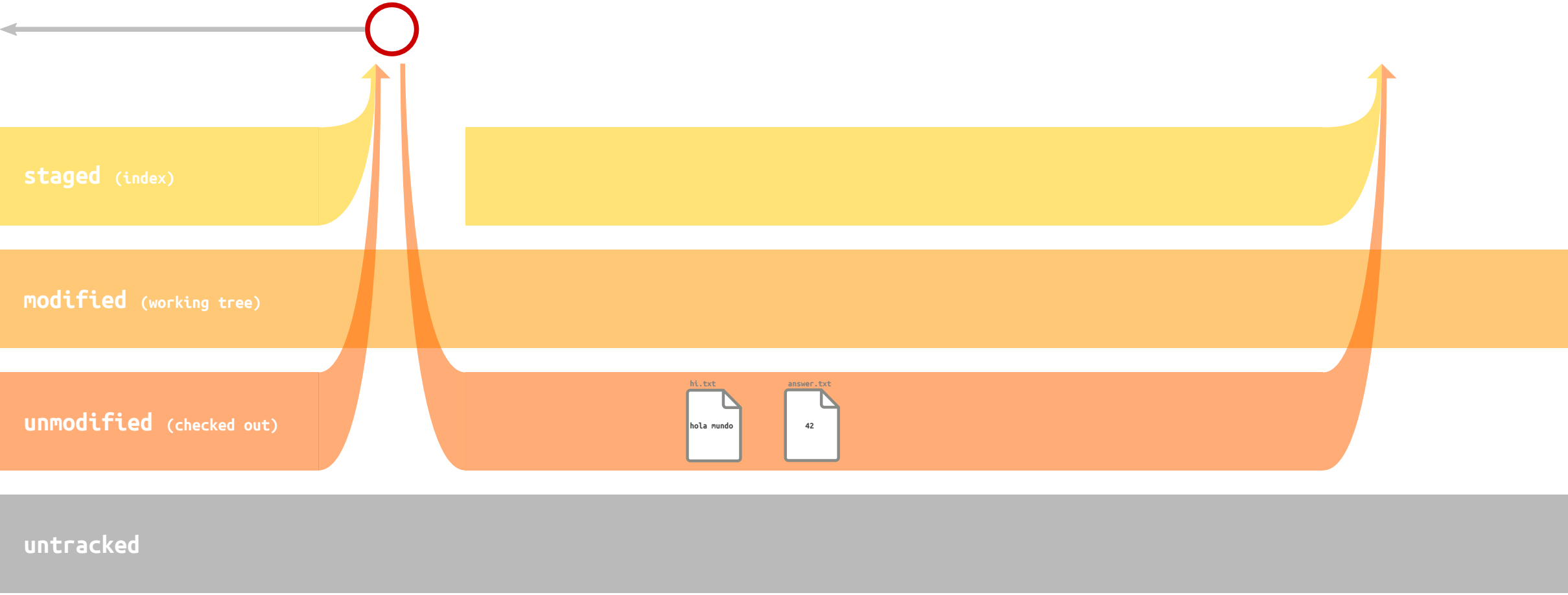
- The 5 (well, 4) file states
  - Or is it 2?
- The circle of life (of a file)
  - From one commit to another
- Anatomy of a commit
  - With all the bloody details
- **Knowing what is happening (from the command line)**
  - **Can be usefull… sometimes**
- Undoing things
  - Can be usefull… too
- Going the extra mile
  - With more complex yet super cool commands

Knowing where you are, from the command line



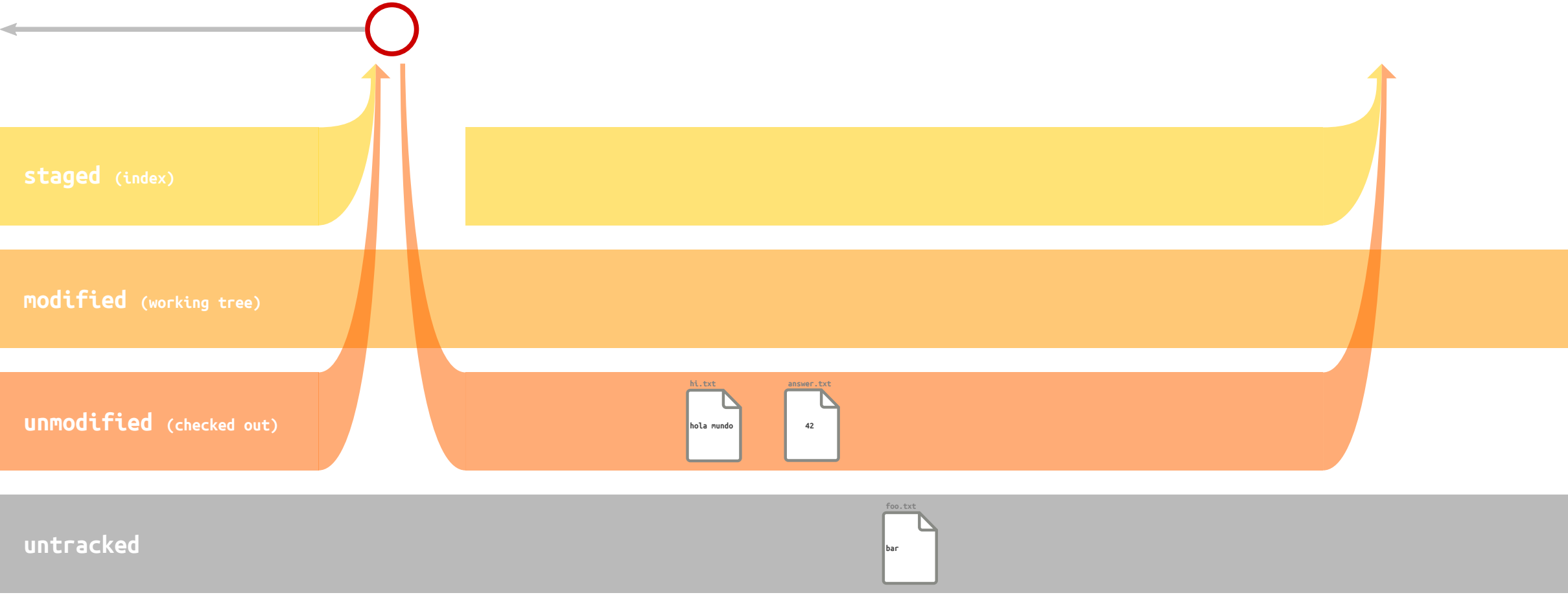
Knowing where you are, from the command line

Use "git status"




```
git status
  On branch main
  nothing to commit, working tree clean
```

Knowing where you are, from the command line



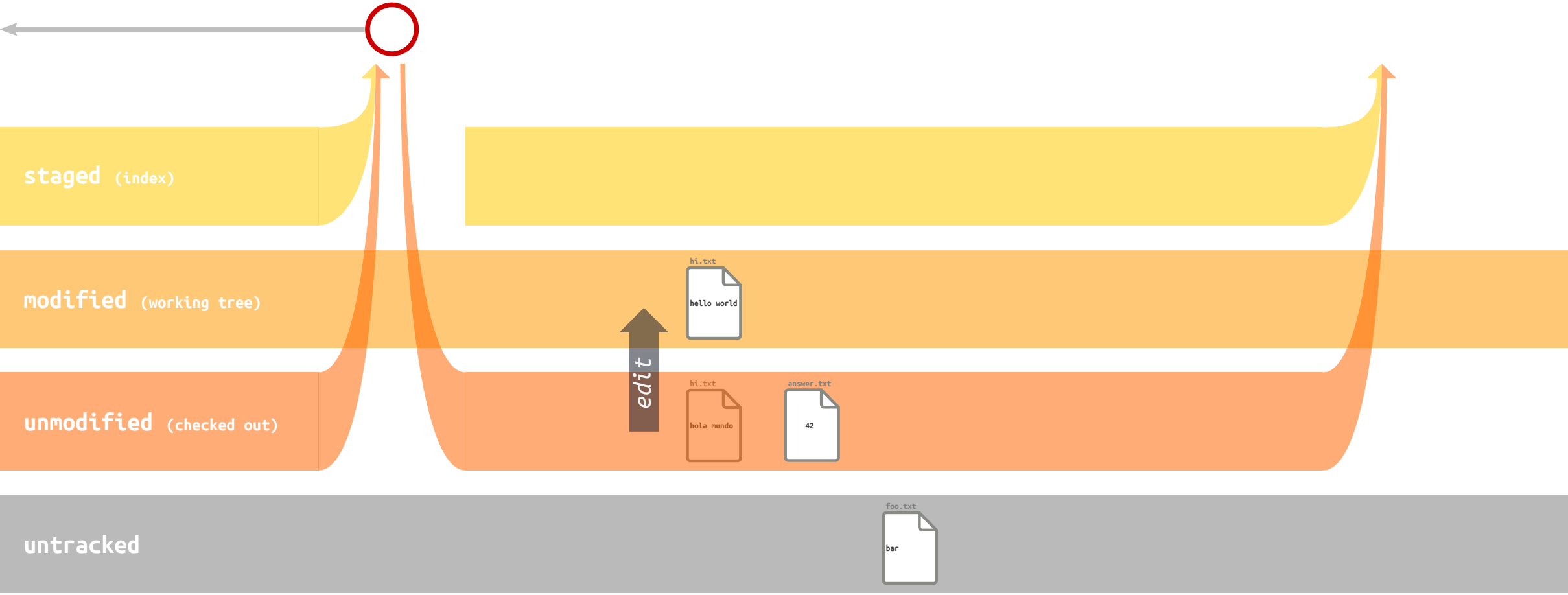

```
echo "bar" > foo.txt
```

Knowing where you are, from the command line


```
git status
On branch main
Untracked files:
 (use "git add <file>..." to include in what will be committed)
    foo.txt
nothing added to commit but untracked files present (use "git add" to track)
```

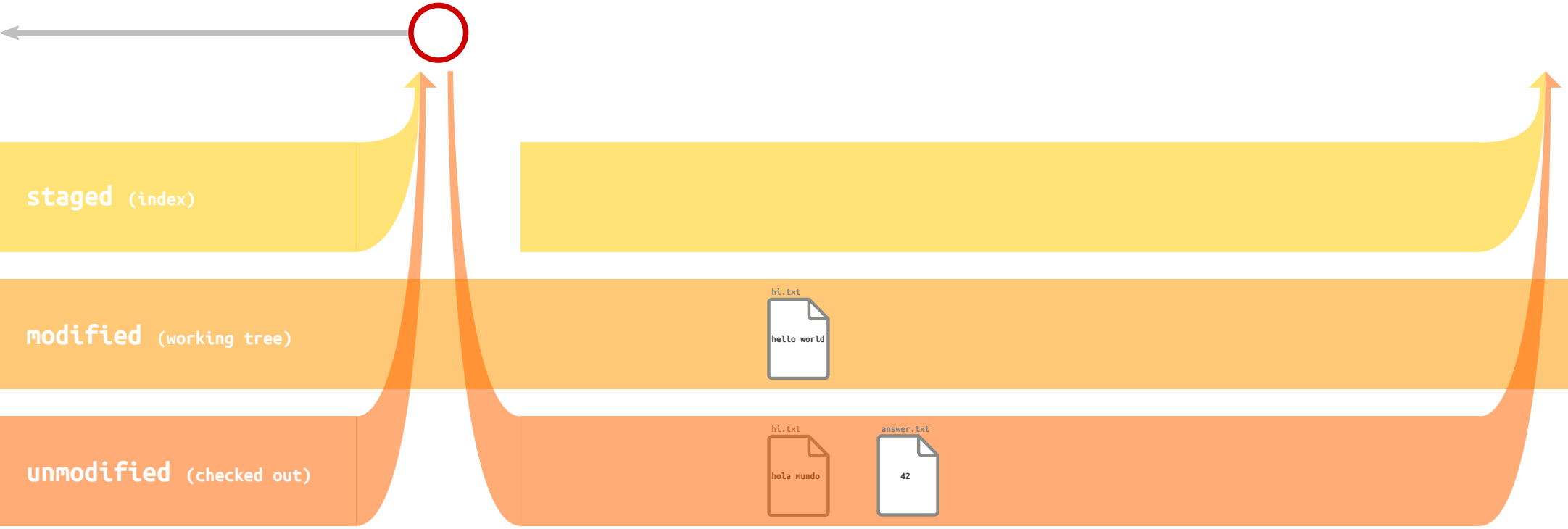
```
git status --short
?? foo.txt
```

Comparing a file across states




```
echo "hello world" > hi.txt
```

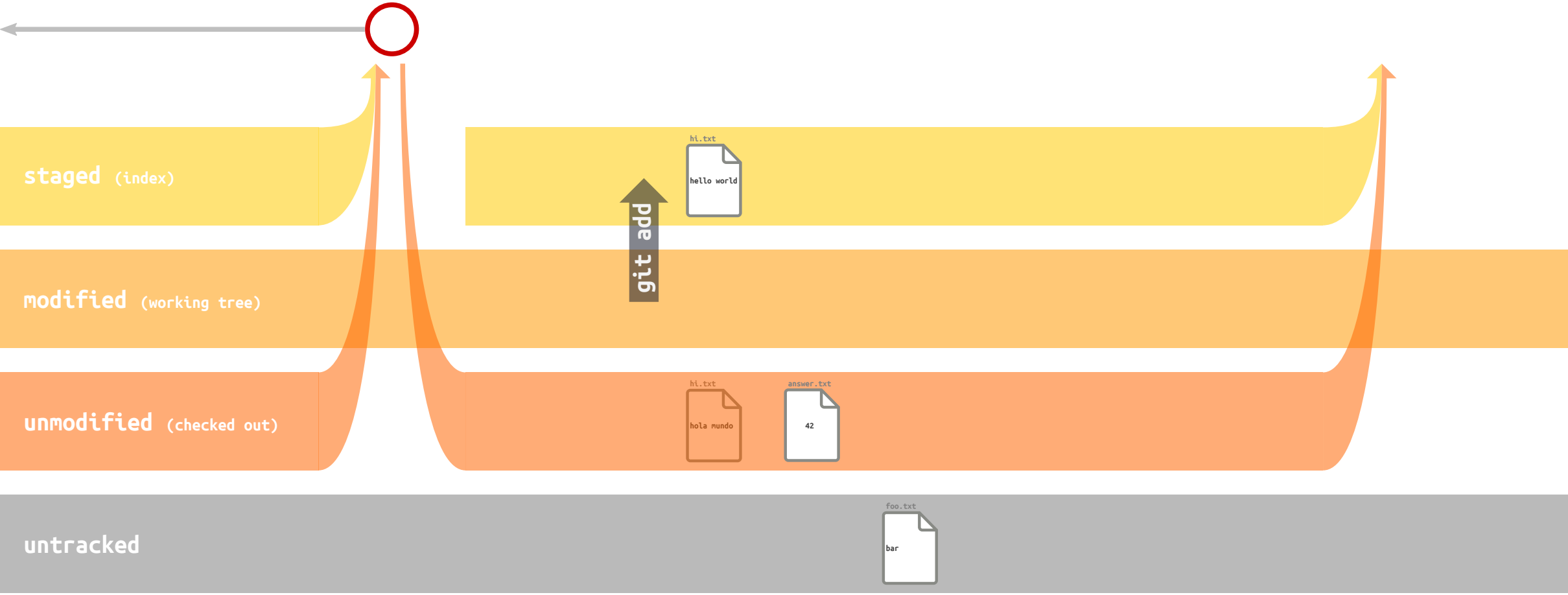
Comparing a file across states




```
git status
On branch main
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)
        modified:   hi.txt
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        foo.txt
no changes added to commit (use "git add" and/or "git commit -a")
```

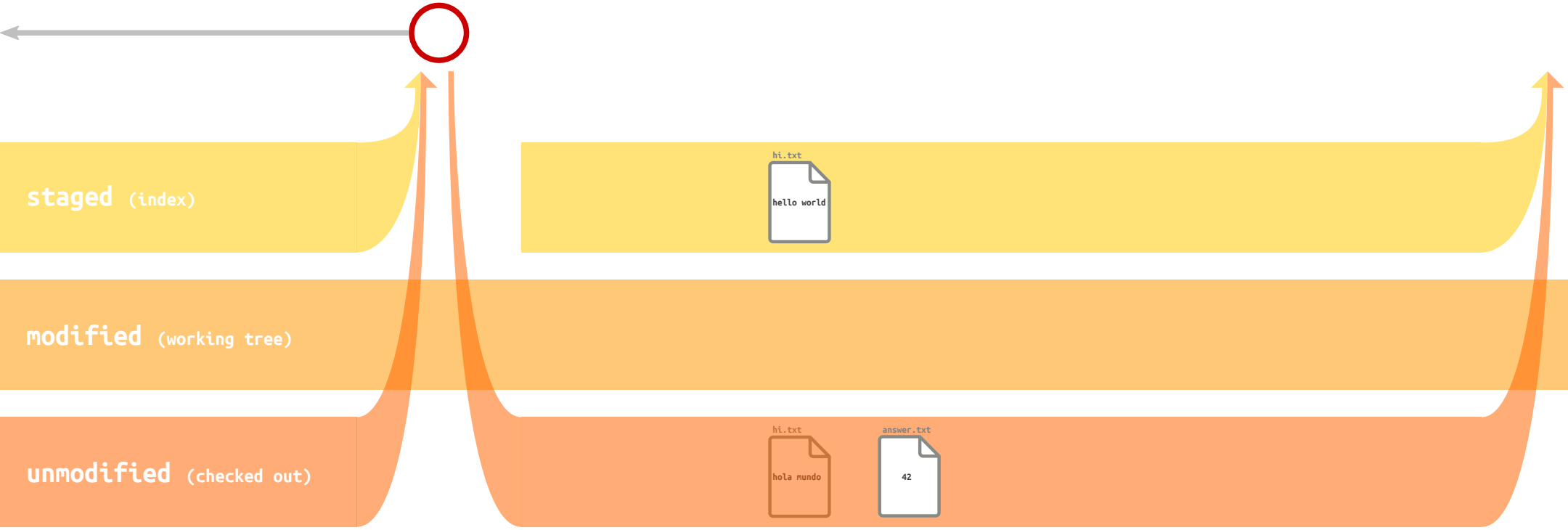
```
git status -s
 M hi.txt
?? foo.txt

(first column represents the index, the second is the working tree)
```

Comparing a file across states




```
git add hi.txt
```

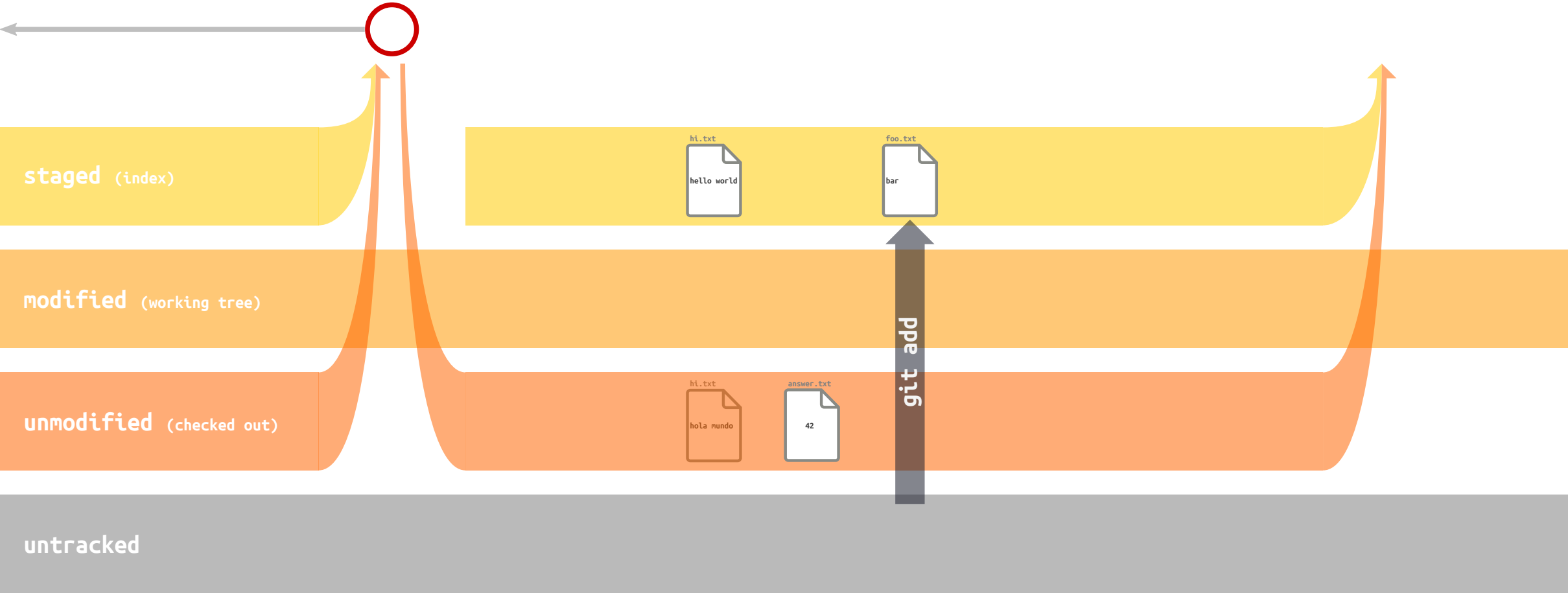
Comparing a file across states




```
git status
  On branch main
  Changes to be committed:
    (use "git reset HEAD <file>..." to unstage)
        modified:   hi.txt
  Untracked files:
    (use "git add <file>..." to include in what will be committed)
        foo.txt
```

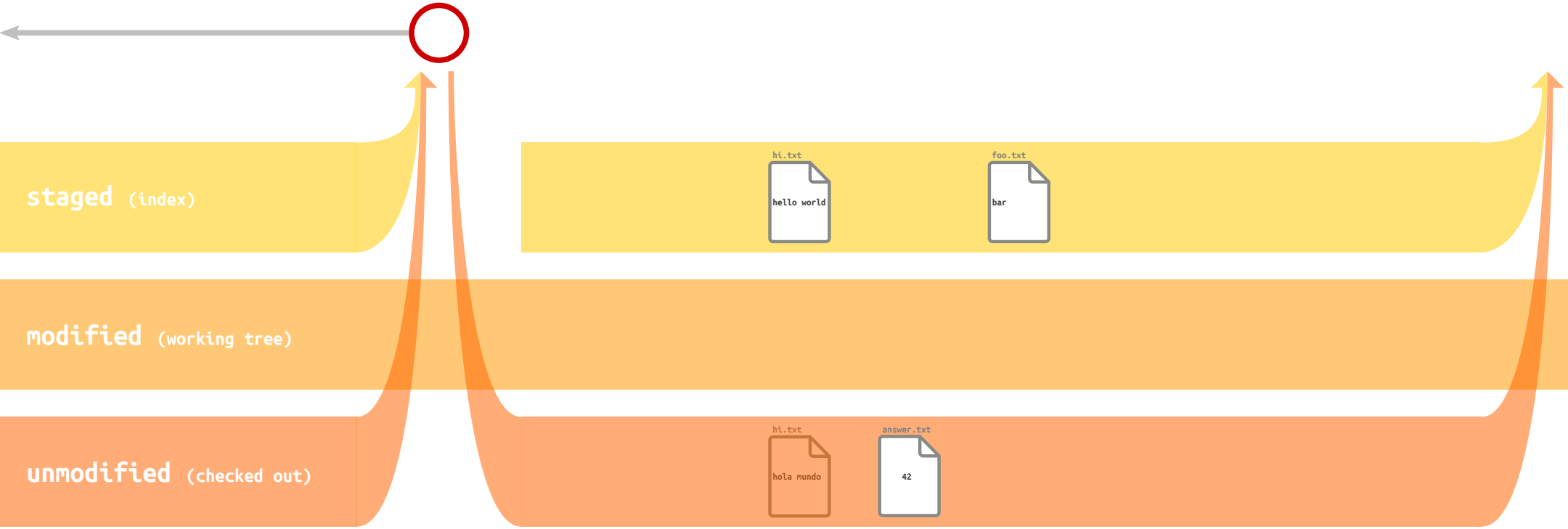
```
git status -s
   M  hi.txt
   ?? foo.txt


(first column represents the index, the second is the working tree)
```

Comparing a file across states




```
git add foo.txt
```

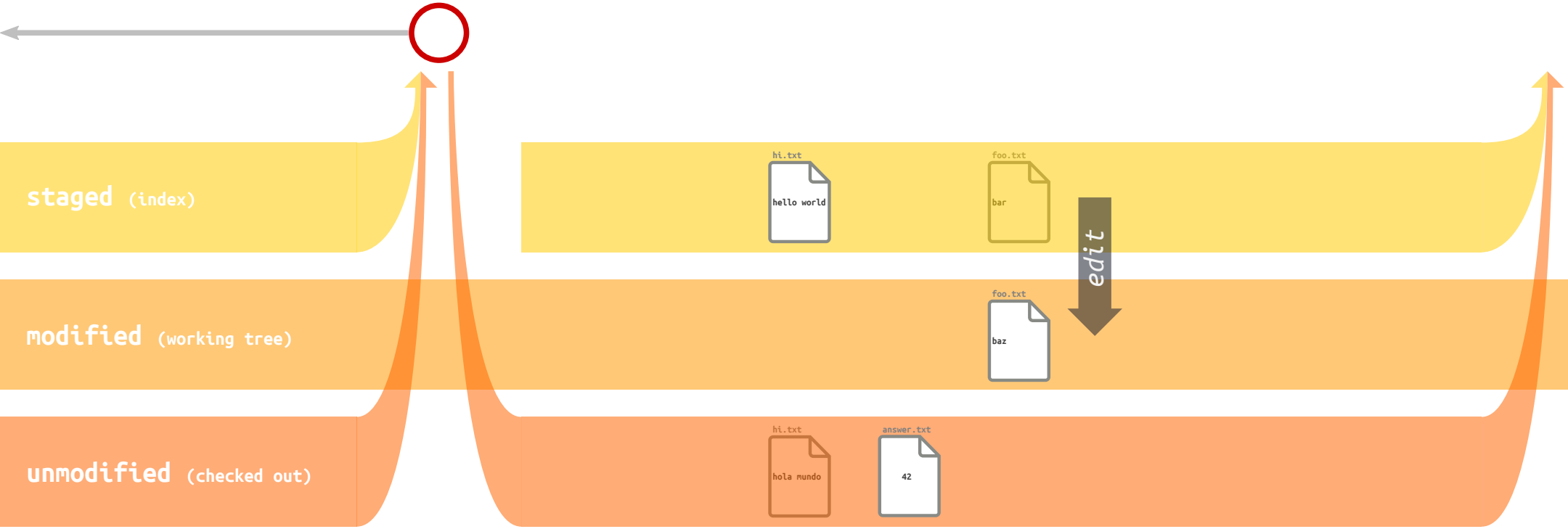
Comparing a file across states




```
git status
  On branch main
  Changes to be committed:
    (use "git reset HEAD <file>..." to unstage)
        new file:   foo.txt
        modified:   hi.txt
```

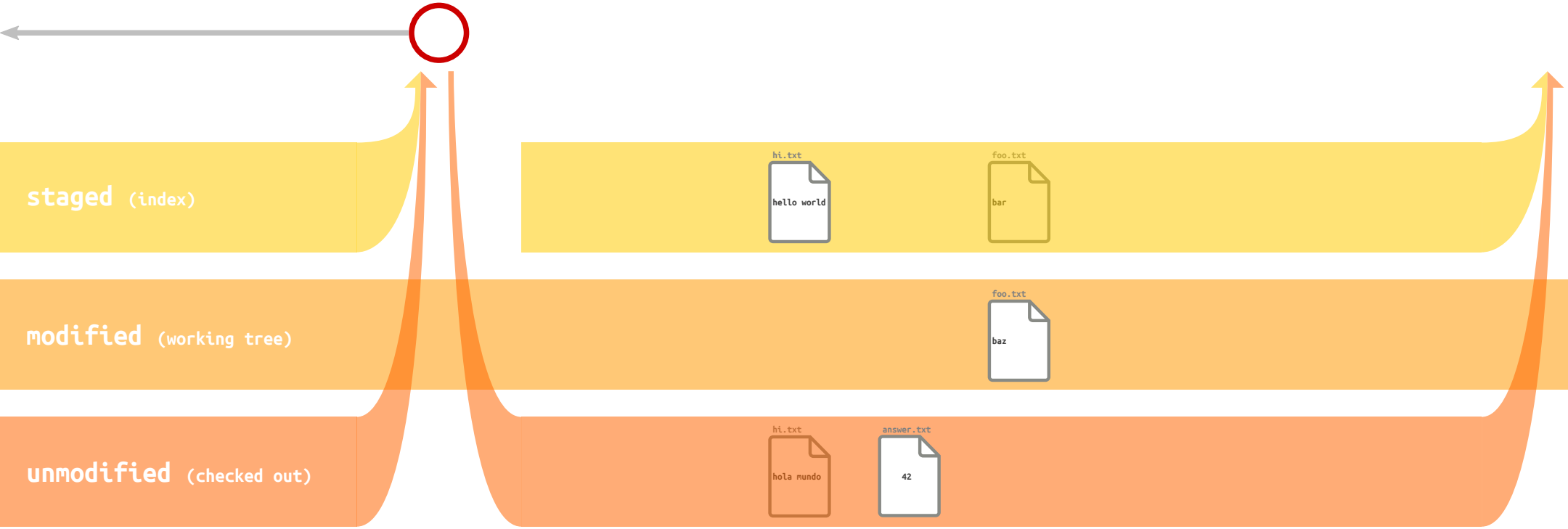
```
git status -s
  A  foo.txt
  M  hi.txt
```

Comparing a file across states




```
echo "baz" > foo.txt
```

Comparing a file across states




```
git status
On branch main
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)
      new file:   foo.txt
      modified:   hi.txt
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)
      modified:   foo.txt
```

```
git status -s
AM foo.txt
M  hi.txt
```

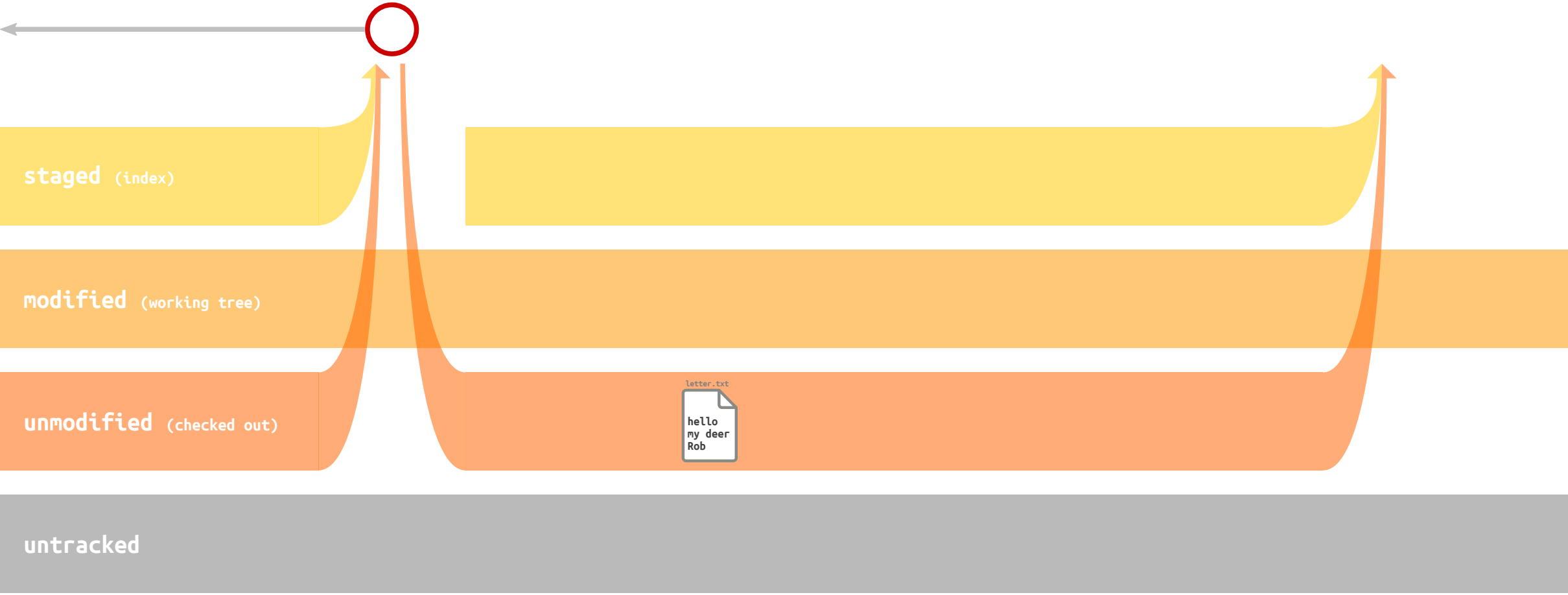
Recap: git status

git status lets you see the **state of each file in the repo** that is neither unmodified, nor ignored.

The verbose version (default) offer a reminder of commands you might need.

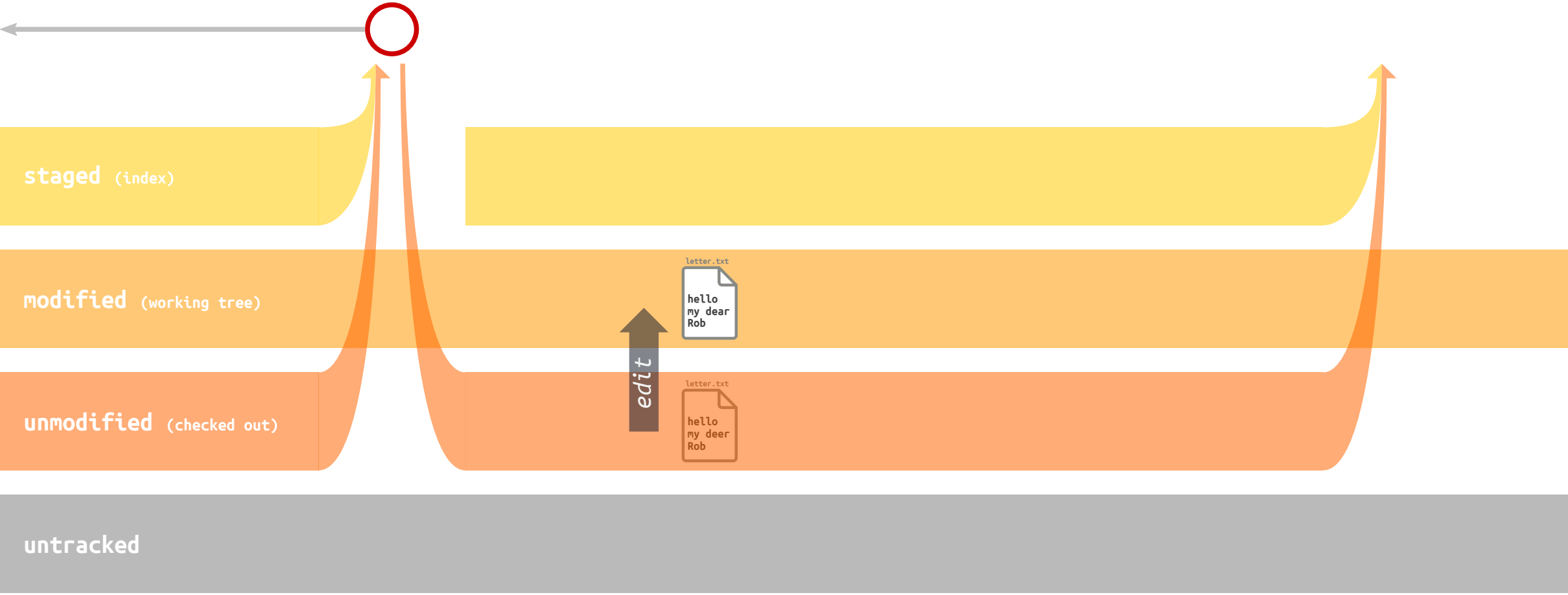
The ***--short*** (or ***-s***) flag is **useful for brevety**.

Comparing a file across states



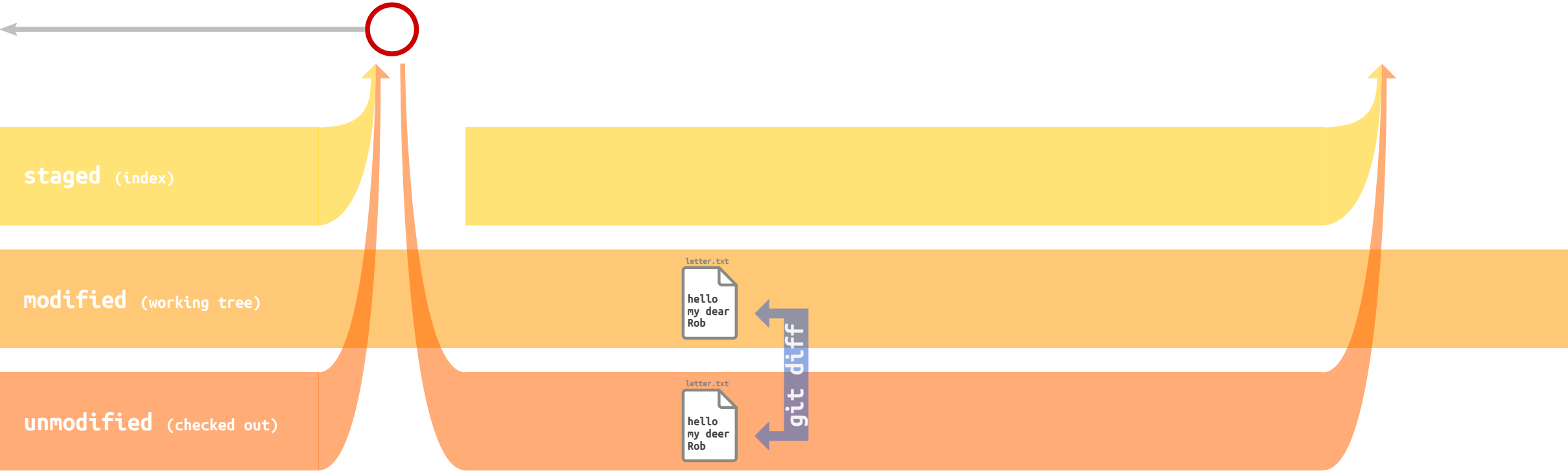

```
echo "hello\nmy deer\nRob" > letter.txt
```

Comparing a file across states



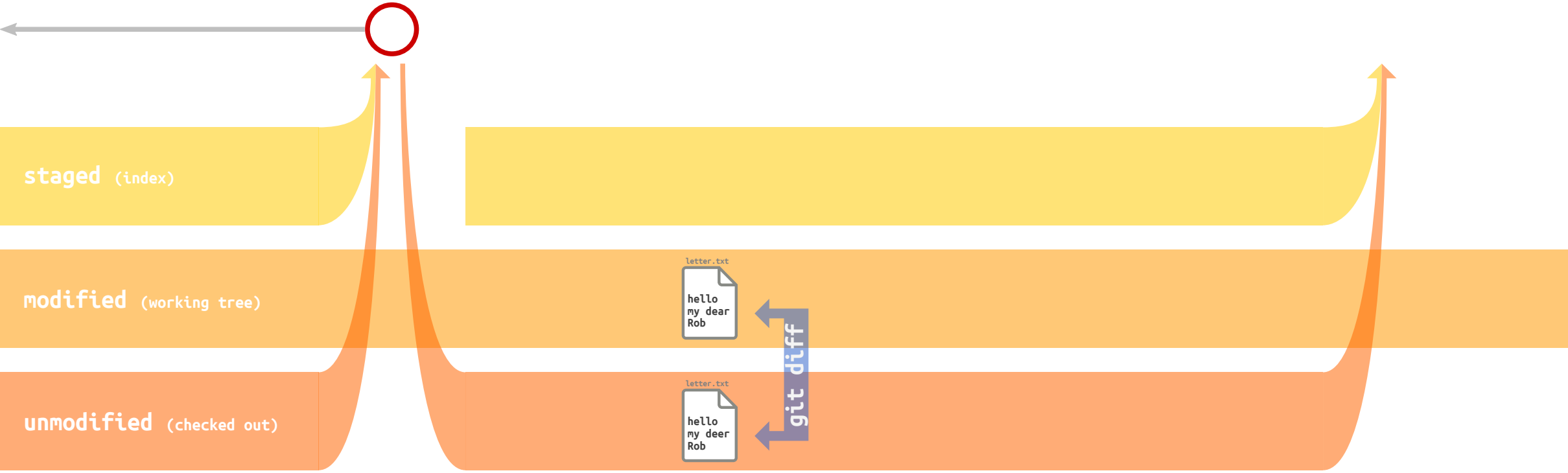

```
echo "hello\nmy dear\nRob" > letter.txt
```

Comparing a file across states




```
git diff letter.txt
  diff --git i/letter.txt w/letter.txt
  index b22fd27..494df86 100644
  --- i/letter.txt
  +++ w/letter.txt
  @@ -1,3 +1,3 @@
   hello
  -my deer
  +my dear
   Rob
```

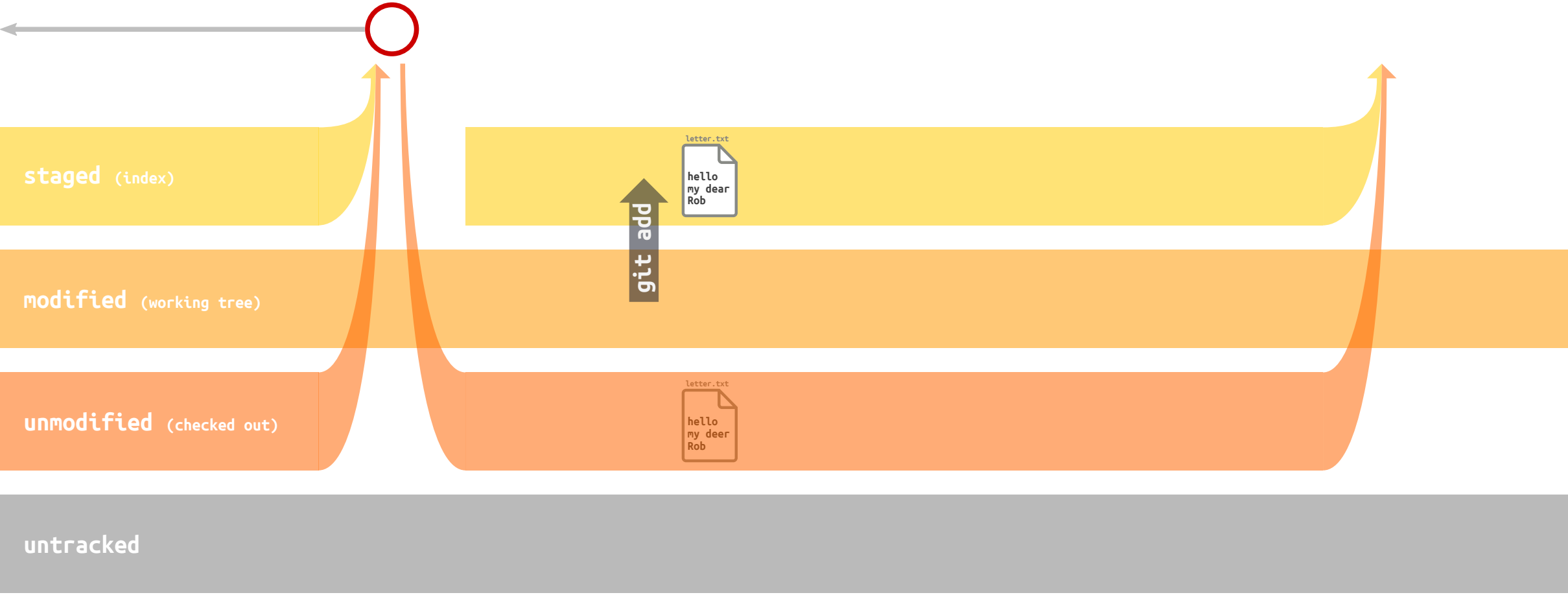
Comparing a file across states




```
git diff letter.txt
diff --git i/letter.txt w/letter.txt
index b22fd27..494df86 100644
--- i/letter.txt
+++ w/letter.txt
@@ -1,3 +1,3 @@
 hello
-my deer
+my dear
 Rob
```

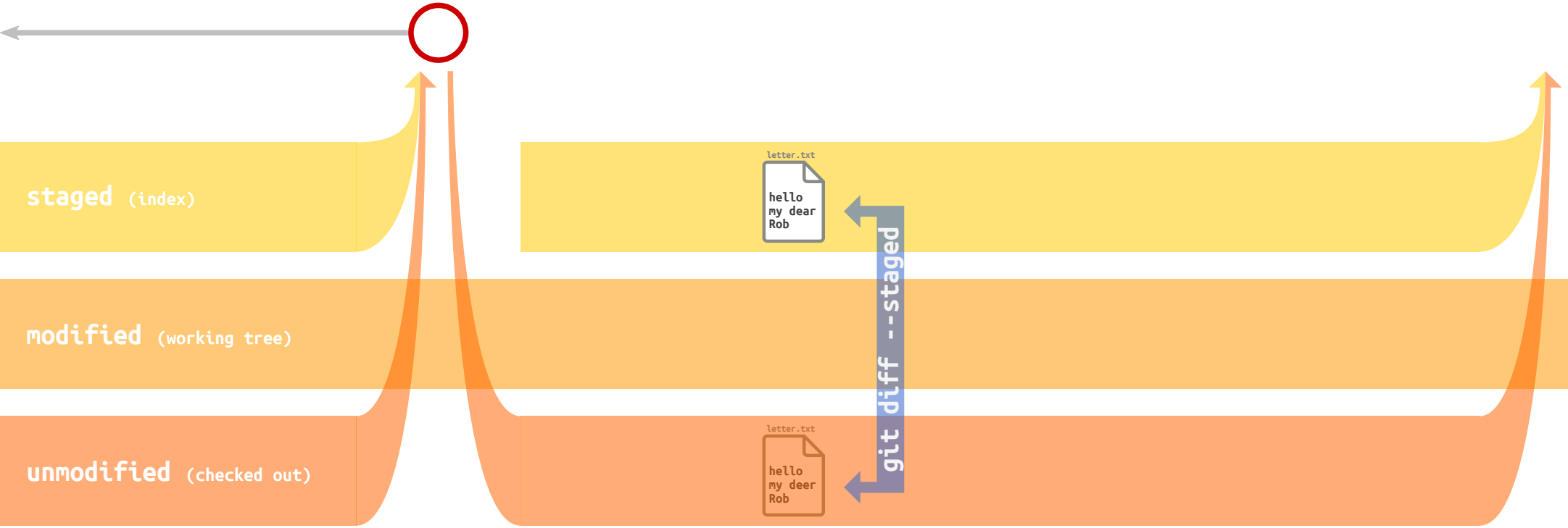
```
git diff --word-diff=color letter.txt
diff --git i/letter.txt w/letter.txt
index b22fd27..494df86 100644
--- i/letter.txt
+++ w/letter.txt
@@ -1,3 +1,3 @@
 hello
 my deerdear
 Rob
```

Comparing a file across states



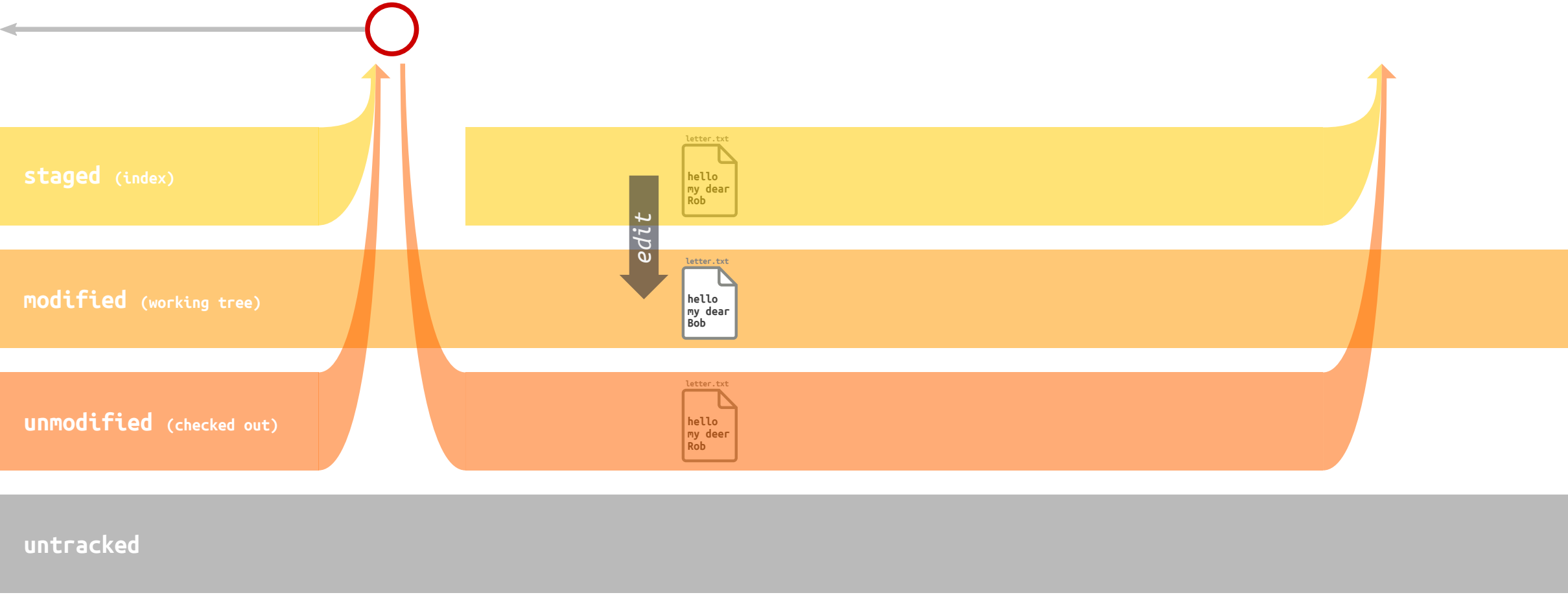

```
git add letter.txt
```

Comparing a file across states



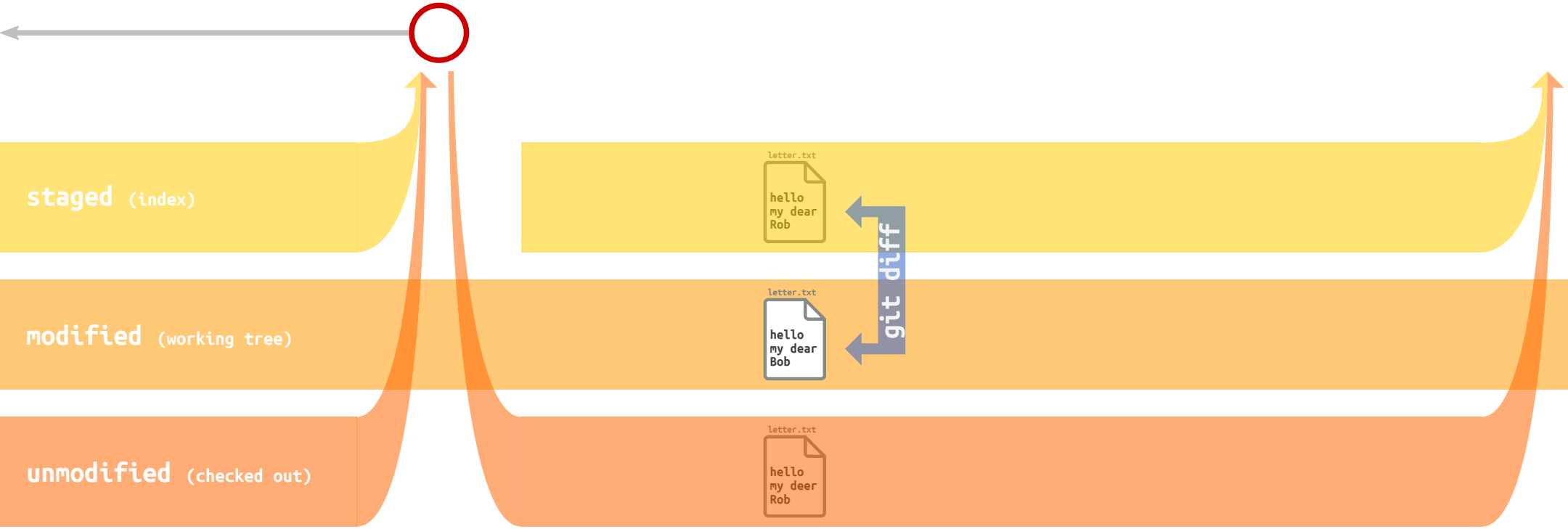

```
git diff --staged letter.txt
  diff --git c/letter.txt i/letter.txt
  index b22fd27..494df86 100644
  --- c/letter.txt
  +++ i/letter.txt
  @@ -1,3 +1,3 @@
   hello
  -my deer
  +my dear
   Rob
```

Comparing a file across states



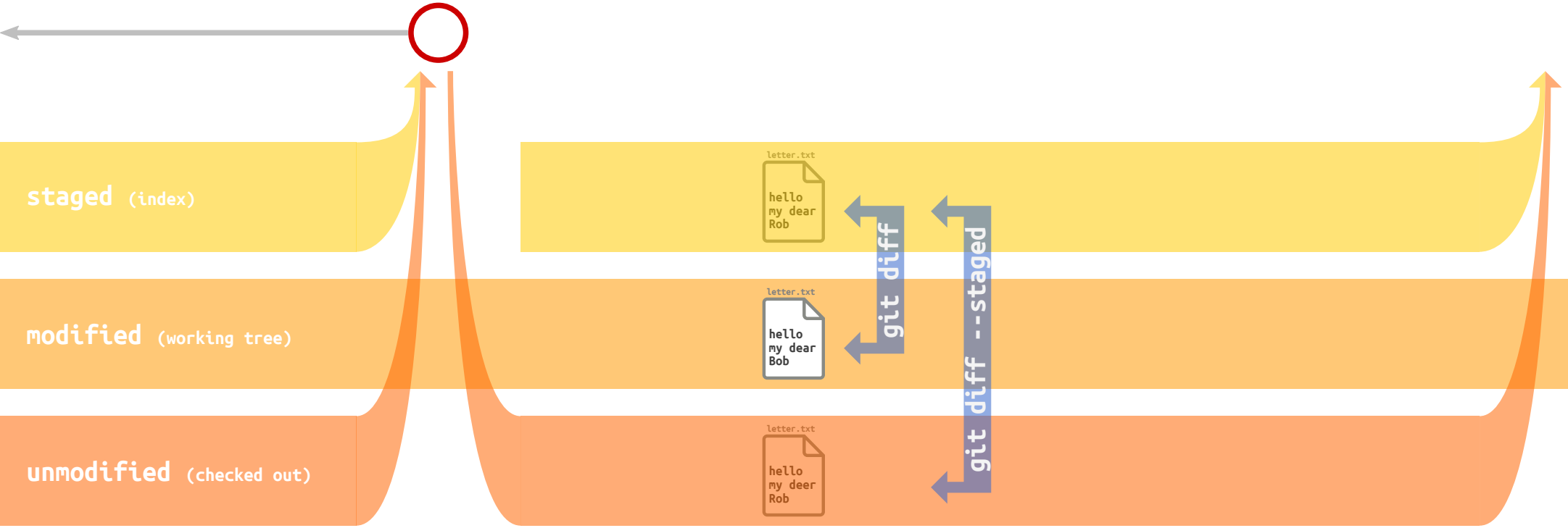

```
echo "hello\nmy dear\nBob" > letter.txt
```

Comparing a file across states



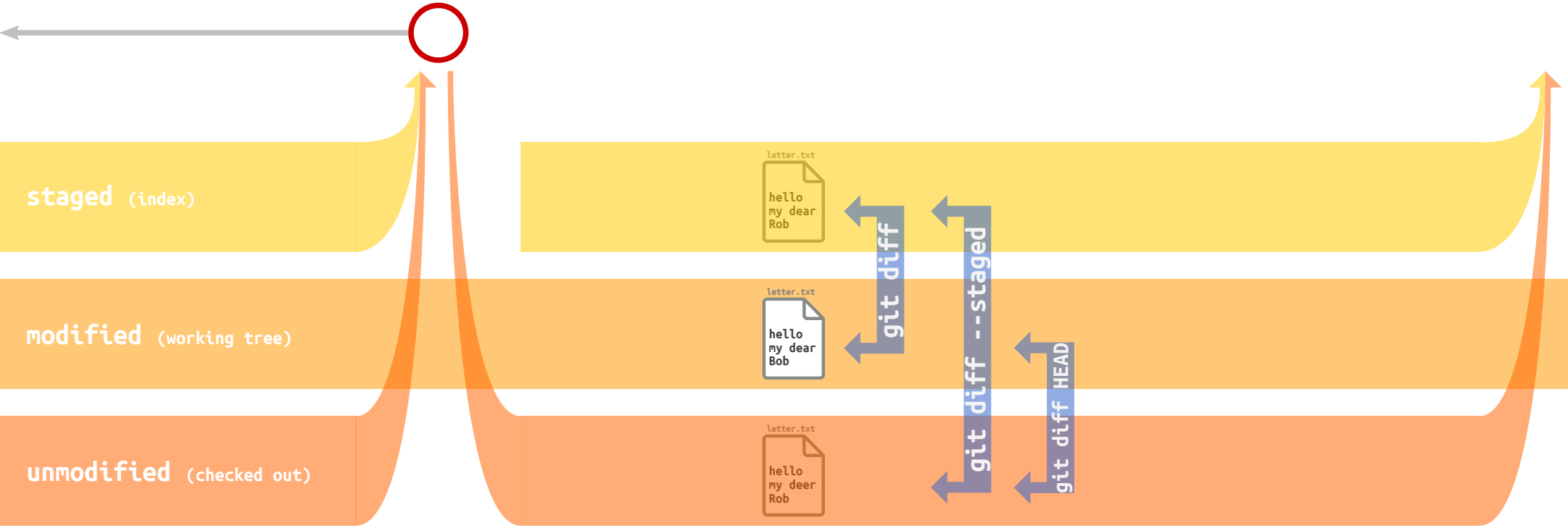

```
git diff letter.txt
  diff --git i/letter.txt w/letter.txt
  index b22fd27..494df86 100644
  --- i/letter.txt
  +++ w/letter.txt
  @@ -1,3 +1,3 @@
   hello
   my dear
  -Rob
  +Bob
```

Comparing a file across states




```
git diff --staged letter.txt
  diff --git c/letter.txt i/letter.txt
  […]
  @@ -1,3 +1,3 @@
   hello
   my dear
  -my deer
  +my dear
   Rob
```

Comparing a file across states




```
git diff HEAD letter.txt
  diff --git c/letter.txt w/letter.txt
  […]
  @@ -1,3 +1,3 @@
   hello
   my dear
  -my deer
  -Rob
  +my dear
  +Bob
```

Recap: git diff

git diff lets you see the **difference in content between two states of a single file.**

It works on multiple files too, by appending all differences.

The default comparison **granularity is the line of code.**

Comparing word to word is also possible.

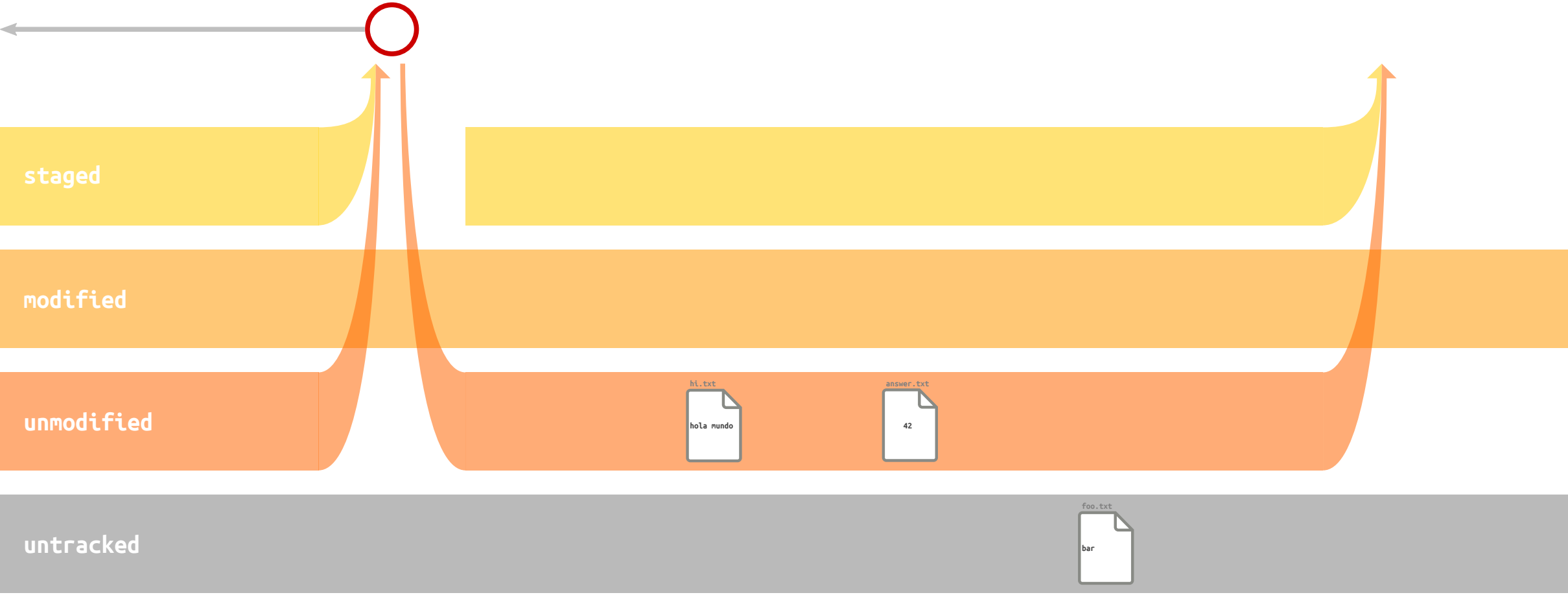
git

What happens at the file-level

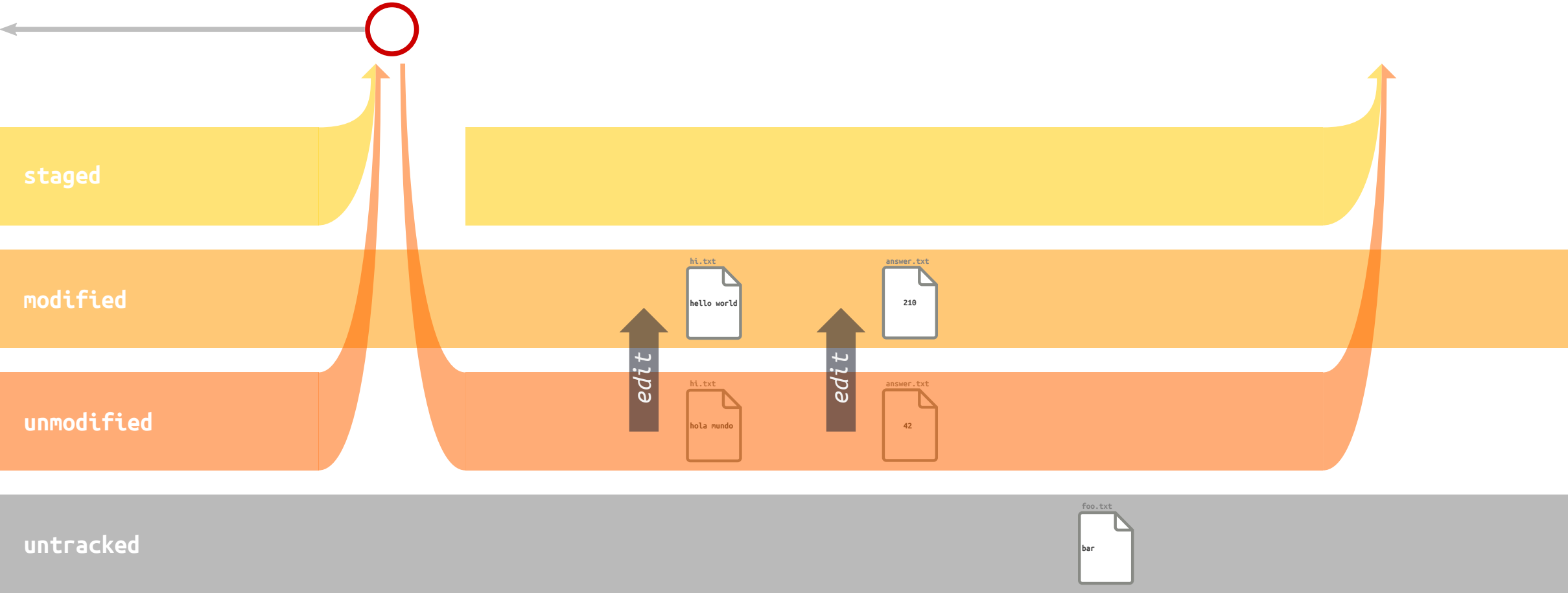
- The 5 (well, 4) file states
  - Or is it 2?
- The circle of life (of a file)
  - From one commit to another
- Anatomy of a commit
  - With all the bloody details
- Knowing what is happening (from the command line)
  - Can be usefull… sometimes
- **Undoing things**
  - **Can be usefull… too**
- Going the extra mile
  - With more complex yet super cool commands

Undoing things

because it can, well, be useful sometimes!



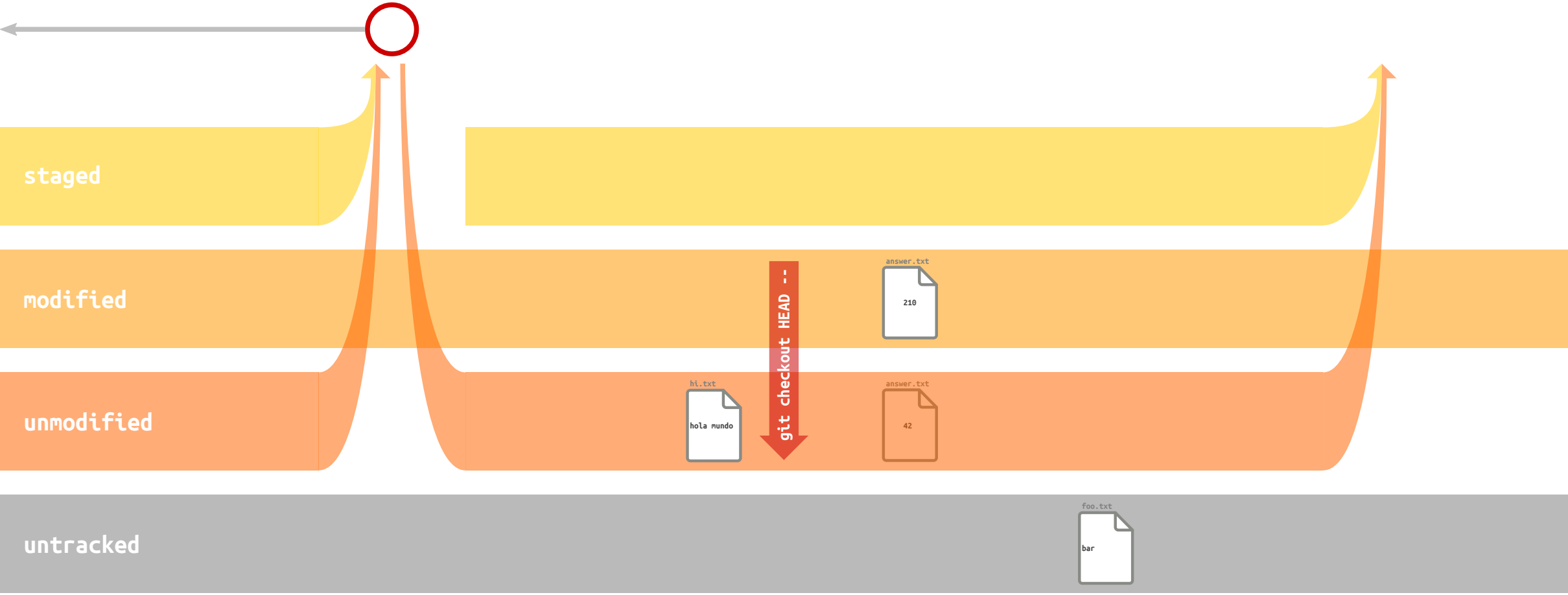
Undo a (non-staged) file edit




```
echo "hello world" > hi.txt
echo "210" > answer.txt
```

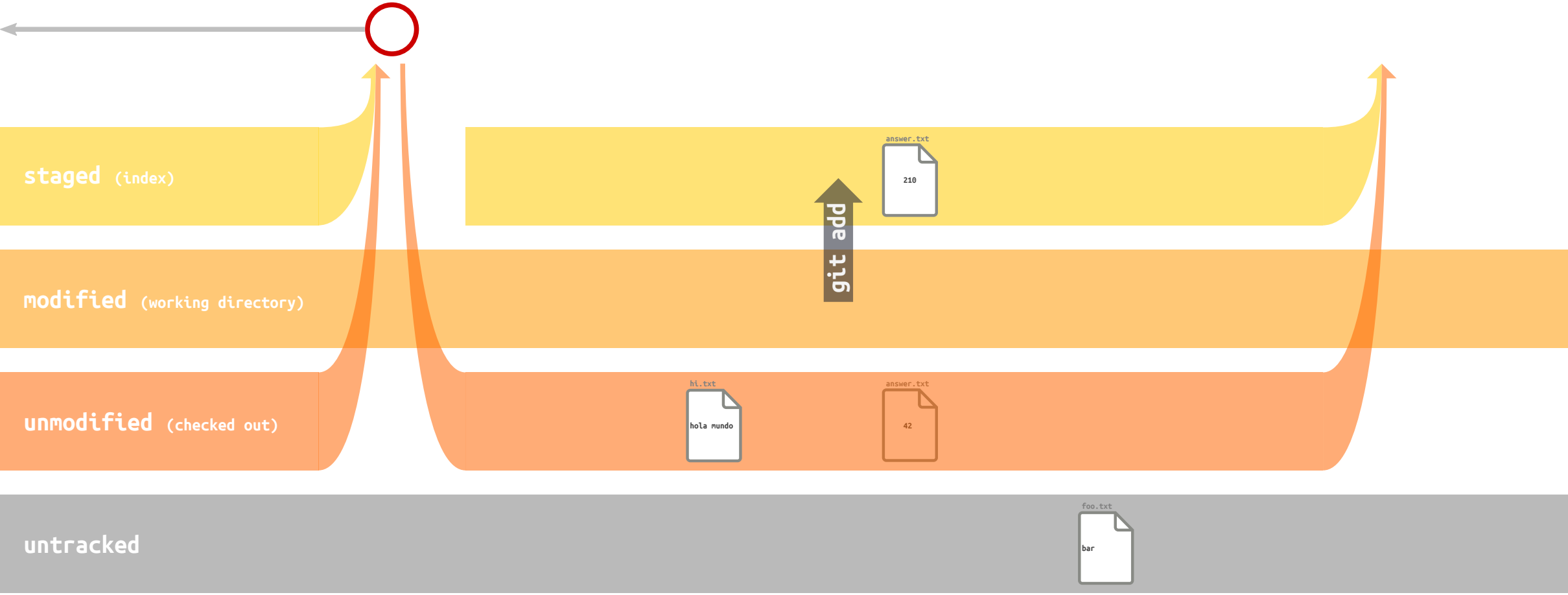
Undo a (non-staged) file edit

CAUTION: you are deleting content, and it's not possible to undo this undo!




```
git checkout HEAD -- hi.txt  CAUTION!
```

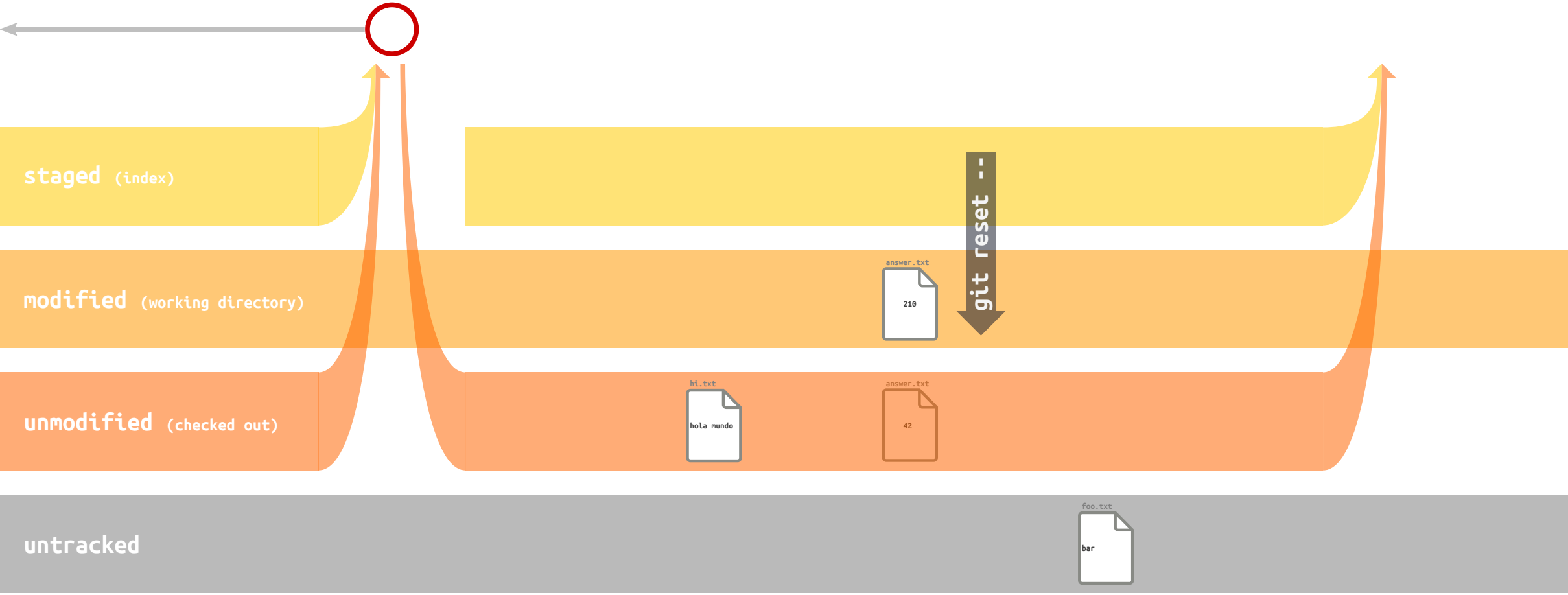
Undo a file staging




```
git add answer.txt
```

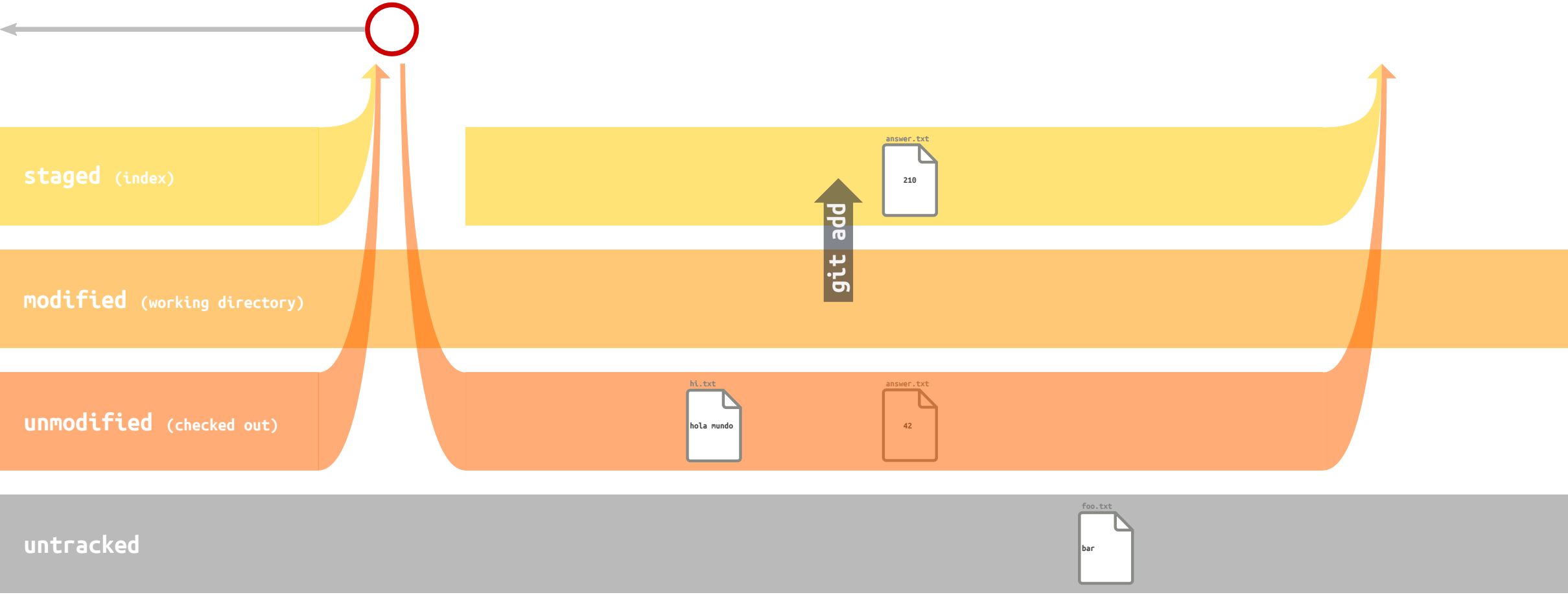
Undo a file staging

git reset is the exact opposit of git add




```
git reset -- answer.txt
```

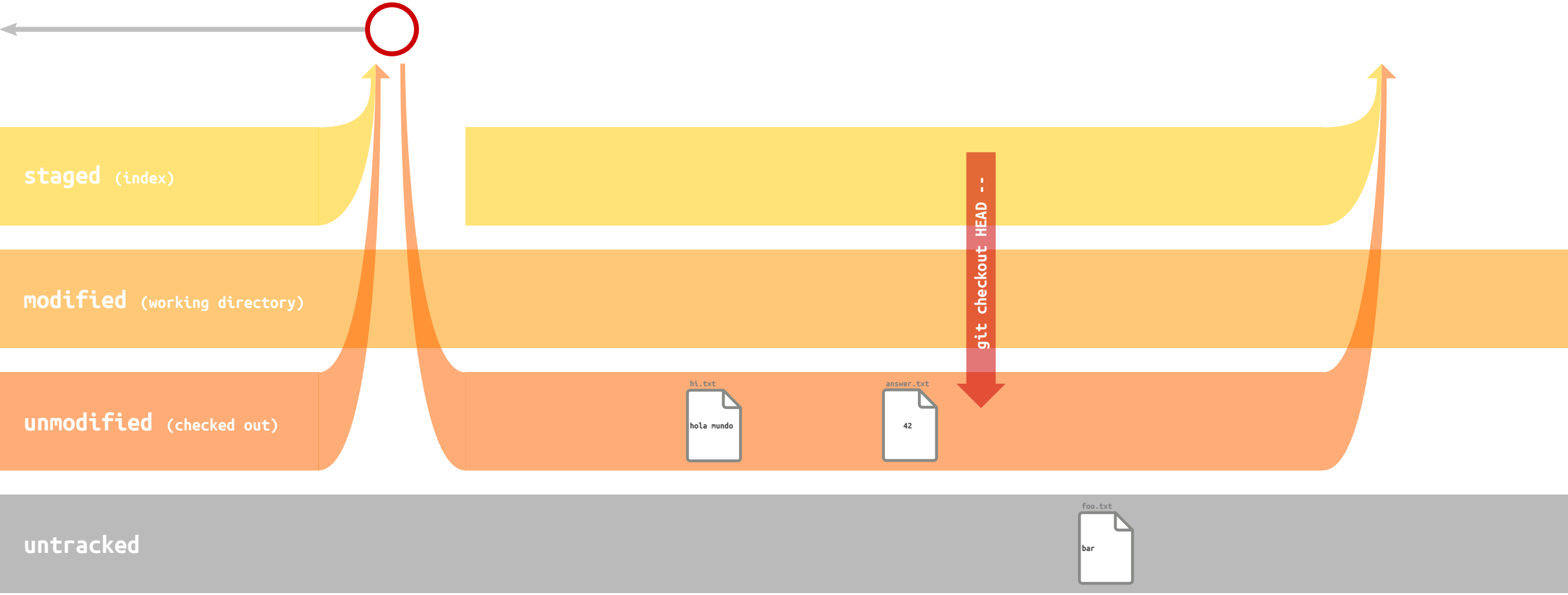
Undo the undo of a file staging: just add it again




```
git add answer.txt
```

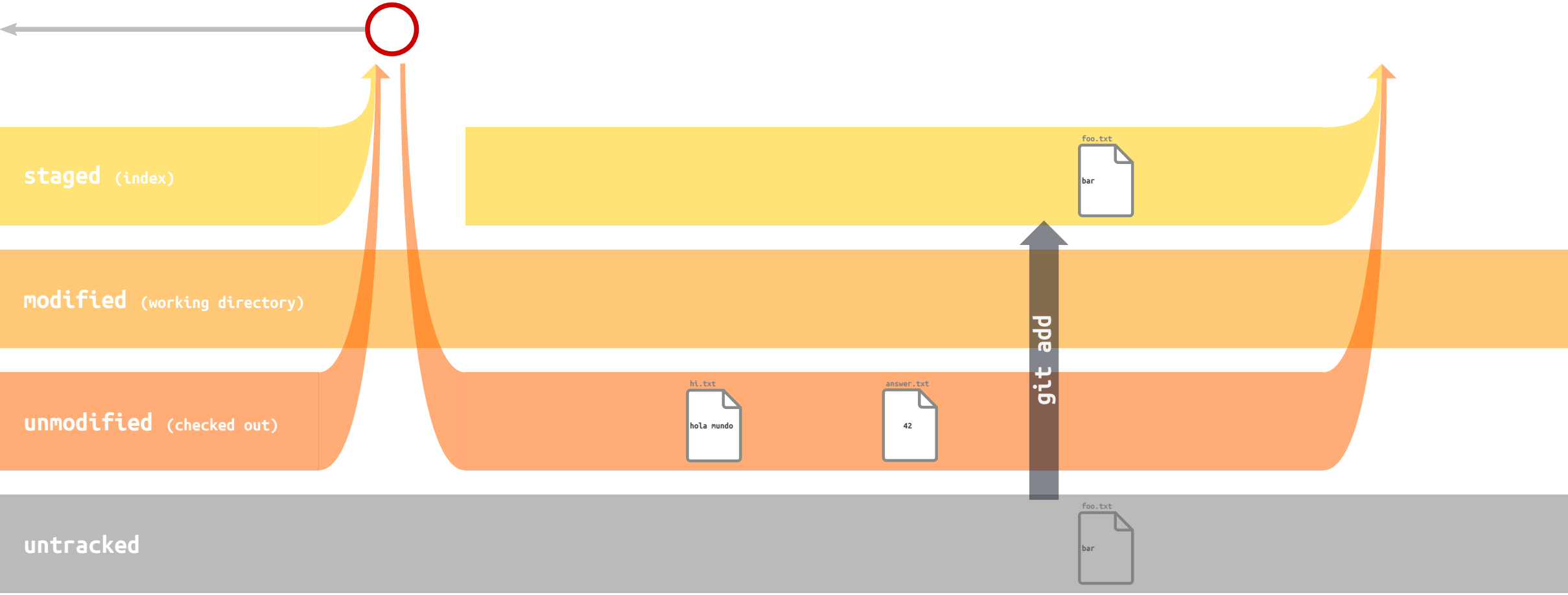
Revert a staged file to its in-the-previous-commit state

CAUTION: you are deleting content, and it's not possible to undo!




```
git checkout HEAD -- answer.txt  CAUTION!
```

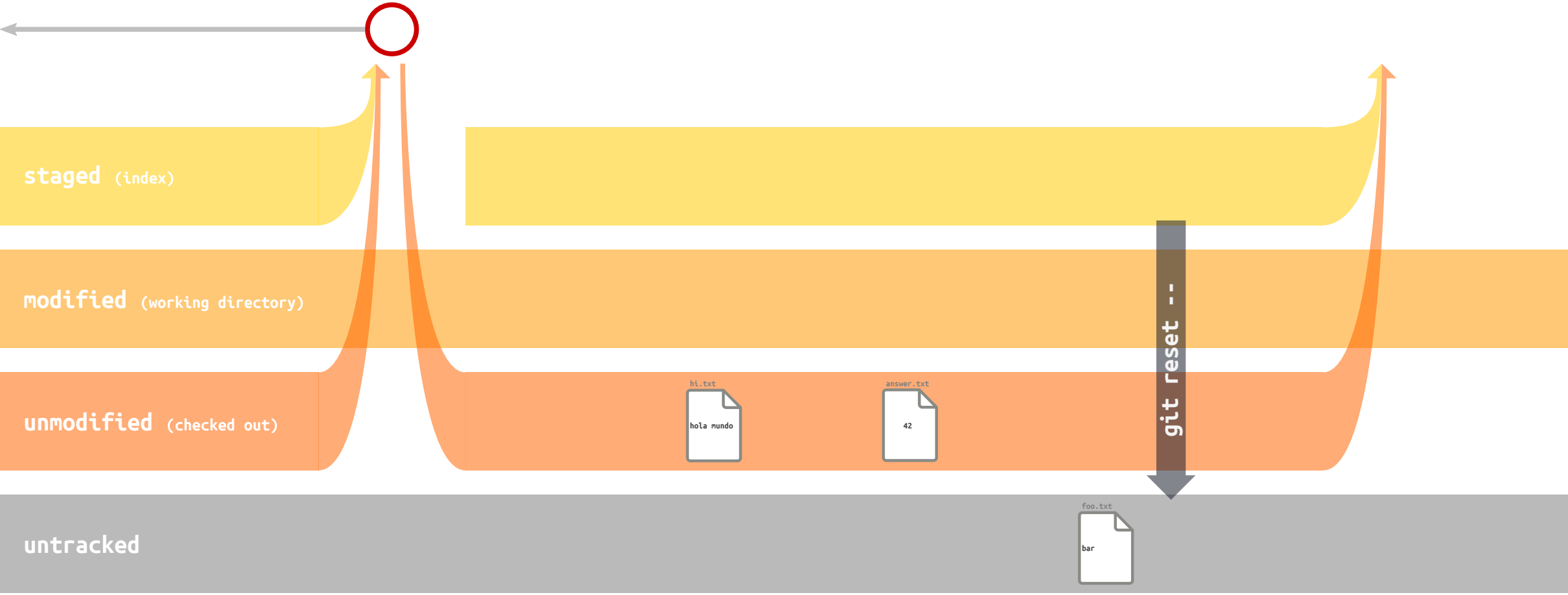
Undo a file addition (staging of a new file)




```
git add foo.txt
```

Undo a file addition (staging of a new file)

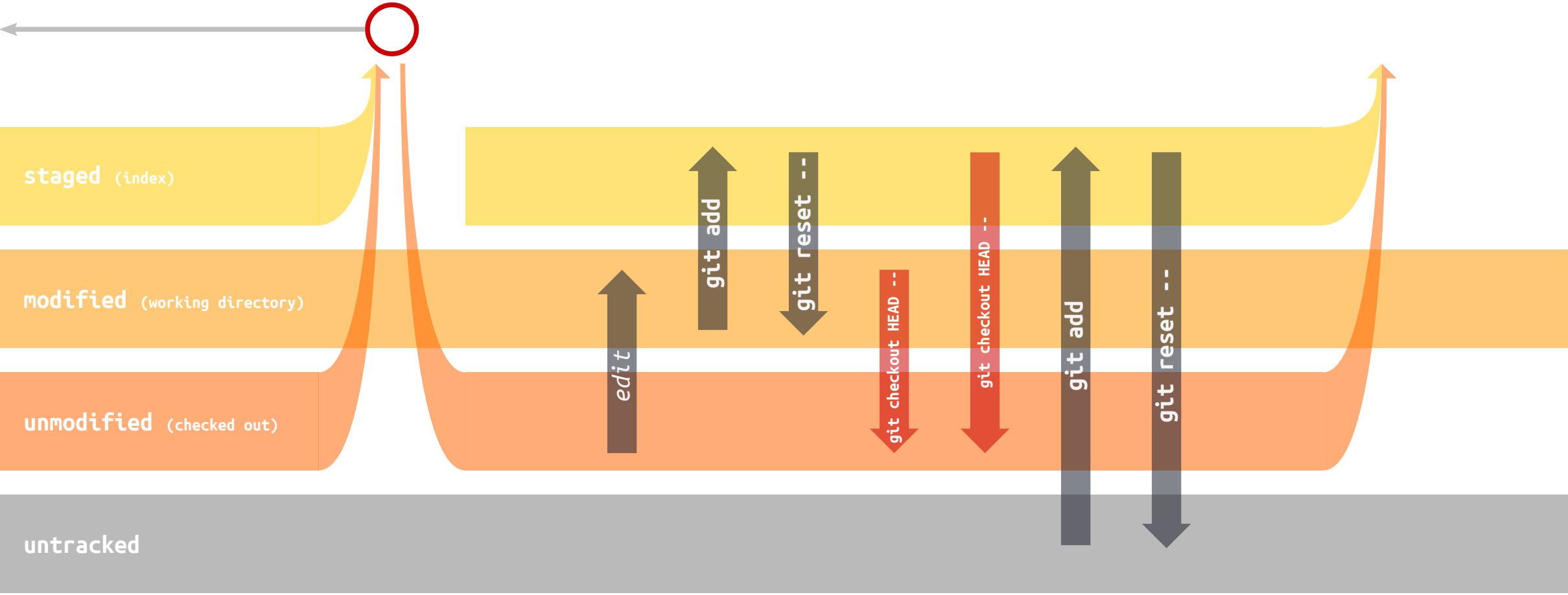
git reset is the exact opposit of git add




```
git reset -- foo.txt
```

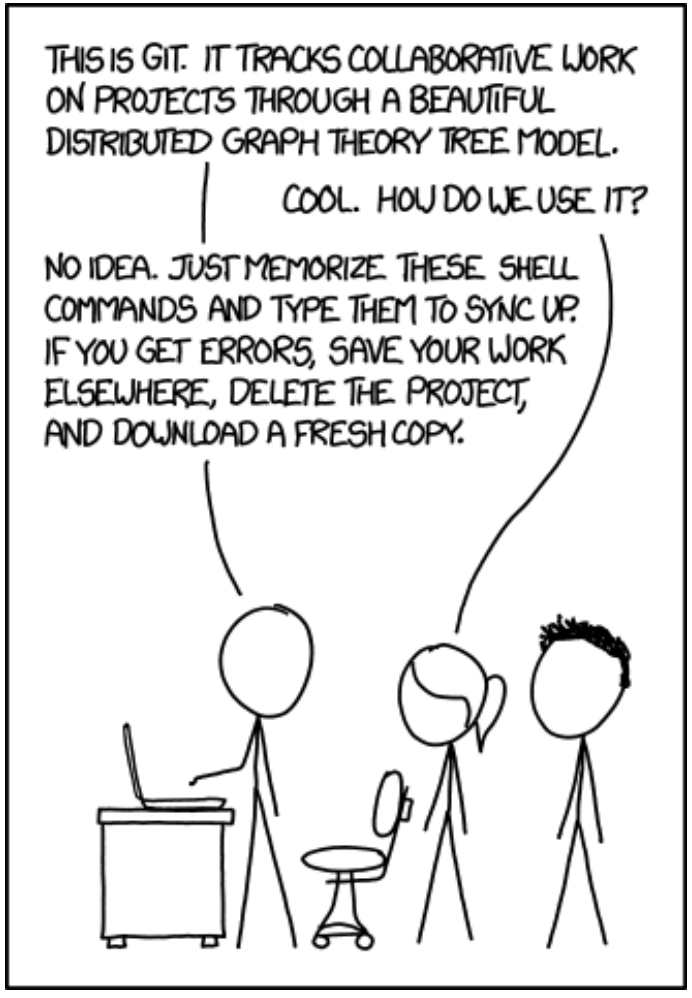
Changing the state of a single file

Let's wrap up

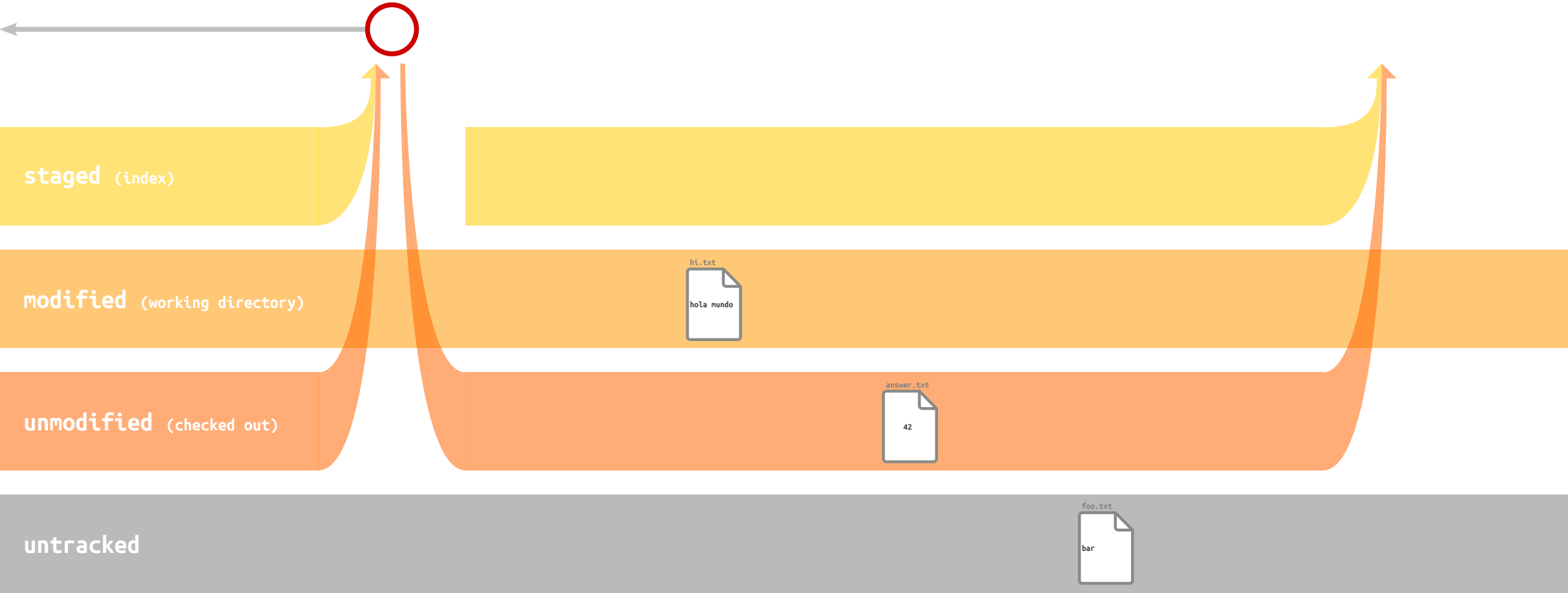


Mandatory xkcd

the other approach

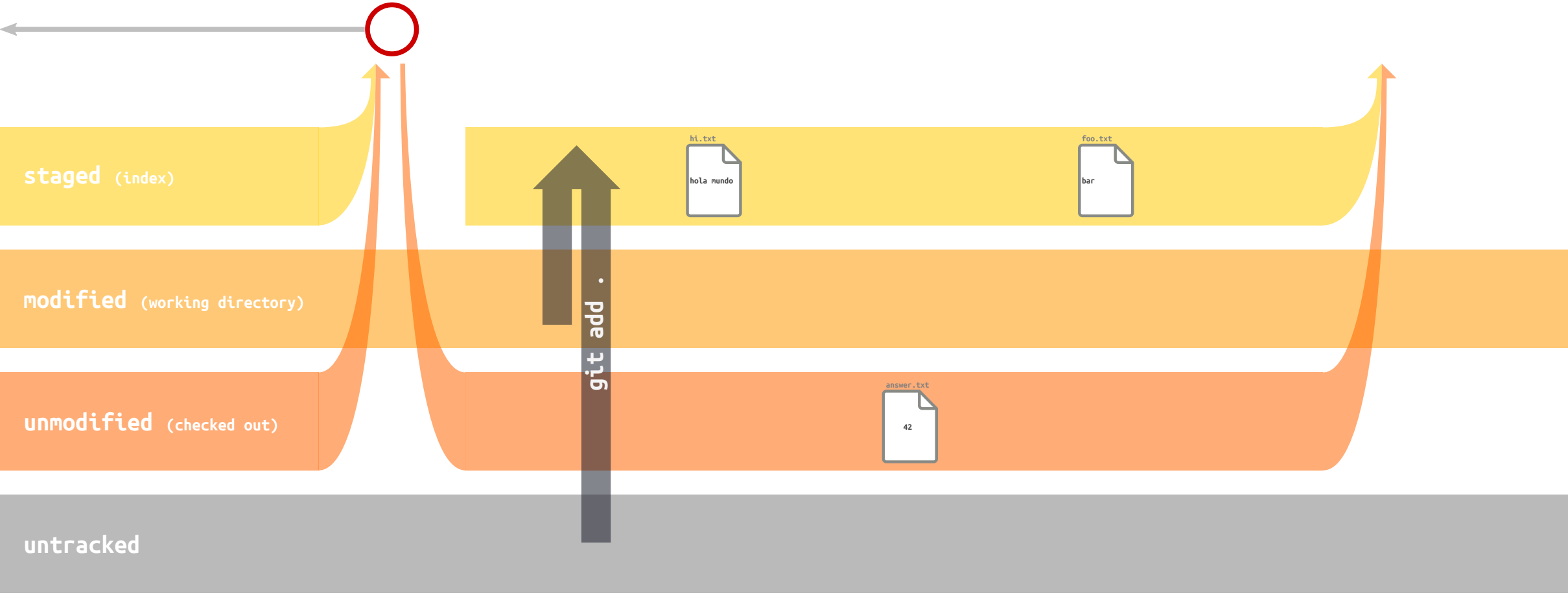


Adding all modified and new (untracked) files



Adding all modified and new (untracked) files

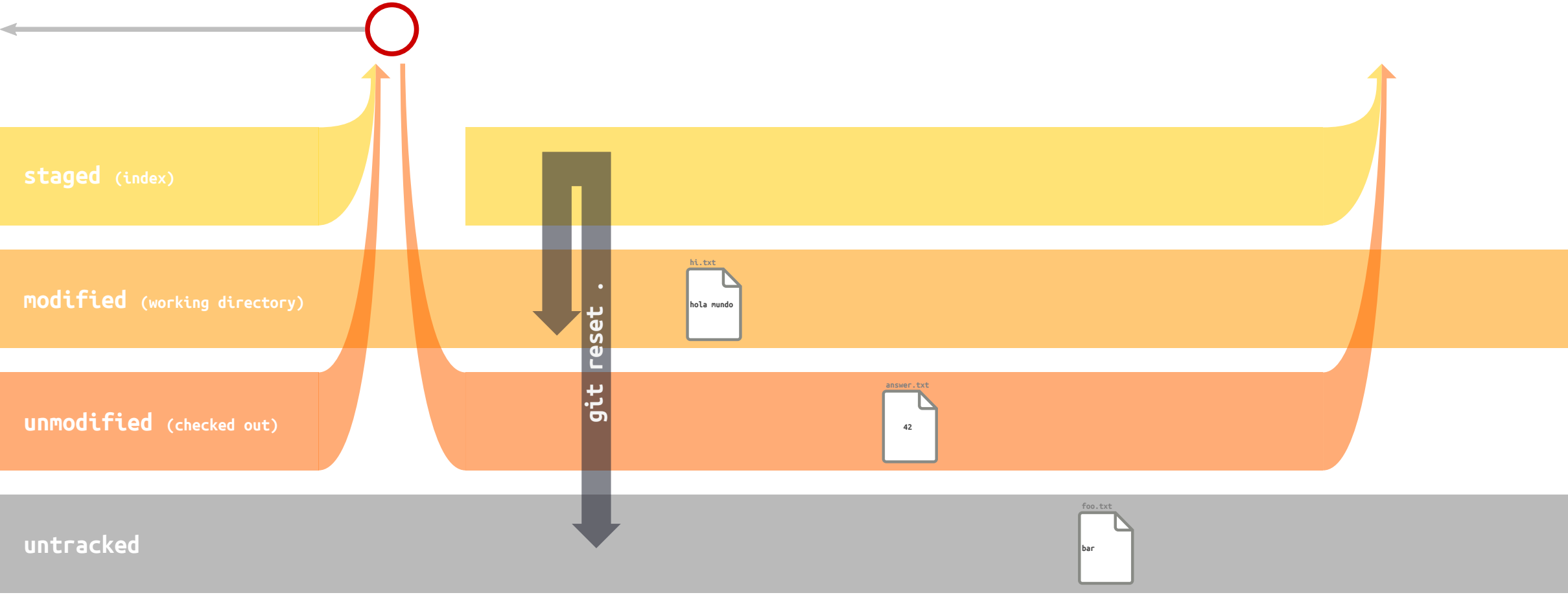
"." means "all files in the directory"




```
git add .
```

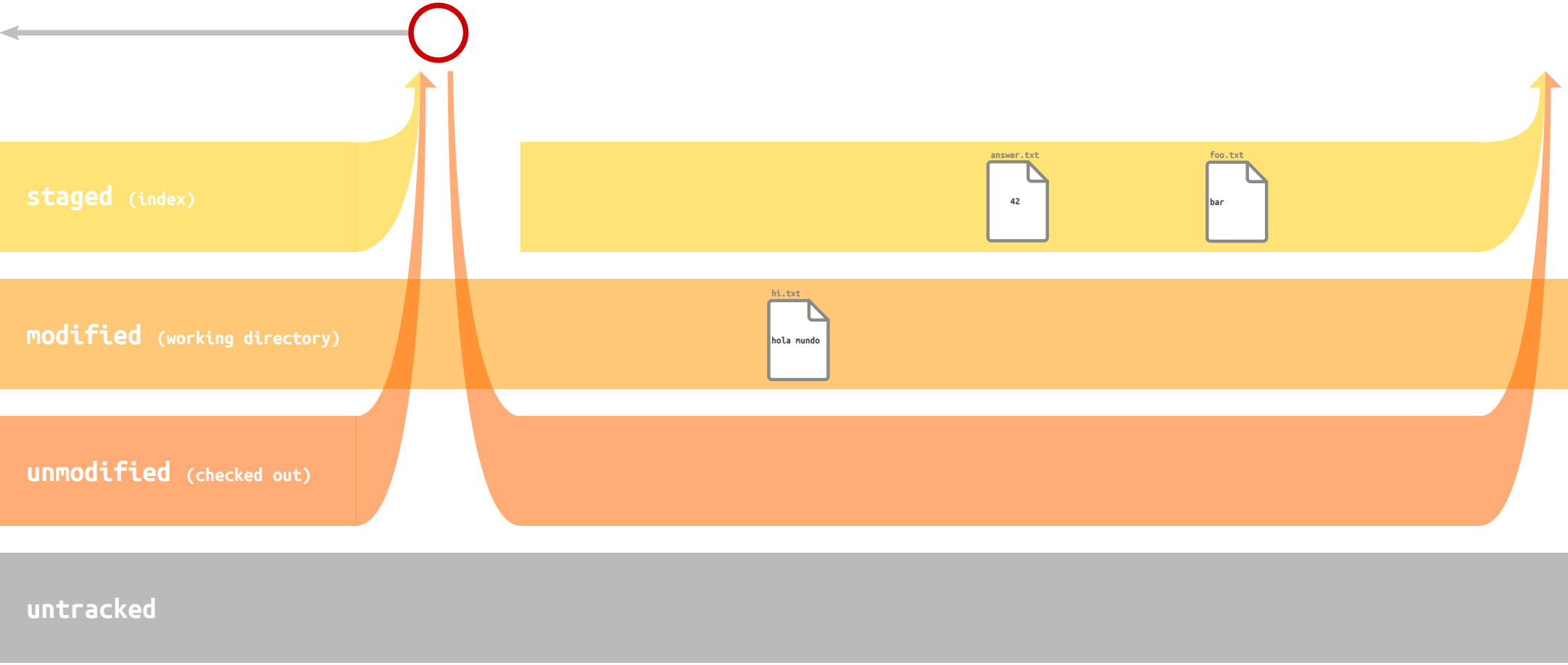
Undo the adding all modified and new (untracked) files

git reset is the exact opposite of git add



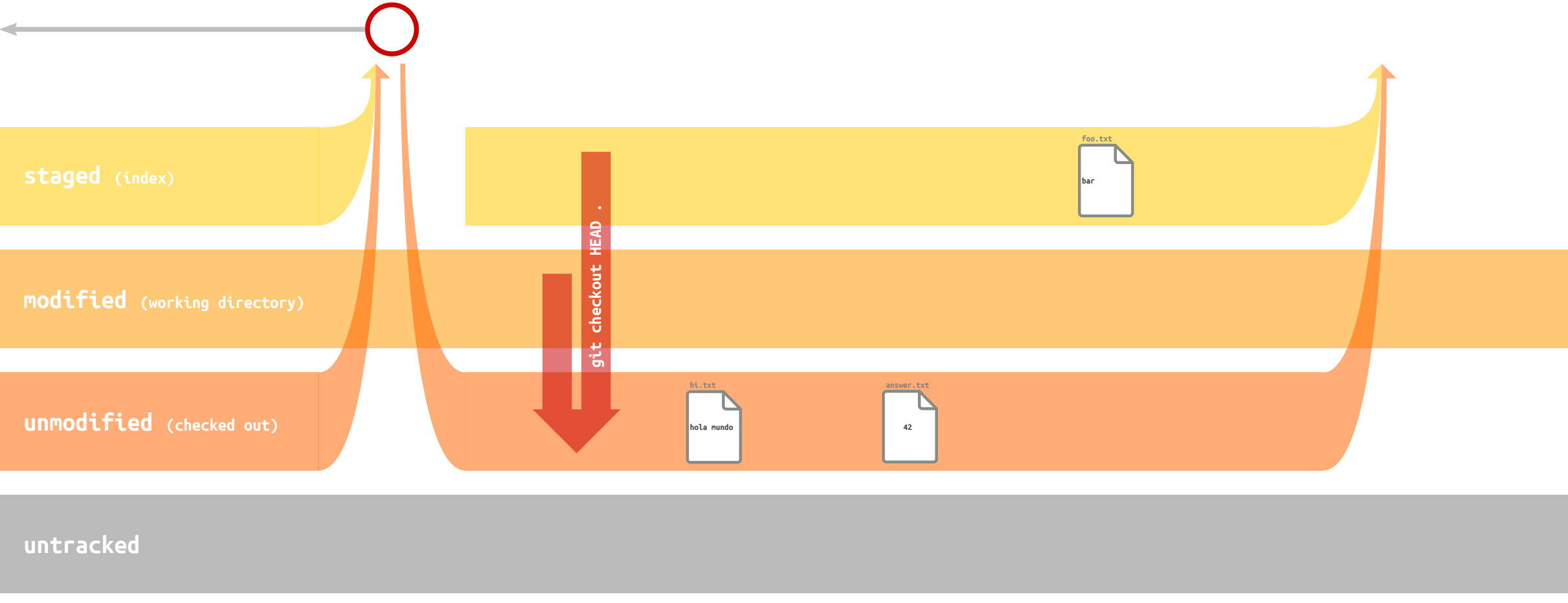

```
git reset .
```

Reverting files to their "in-the-previous-commit" state



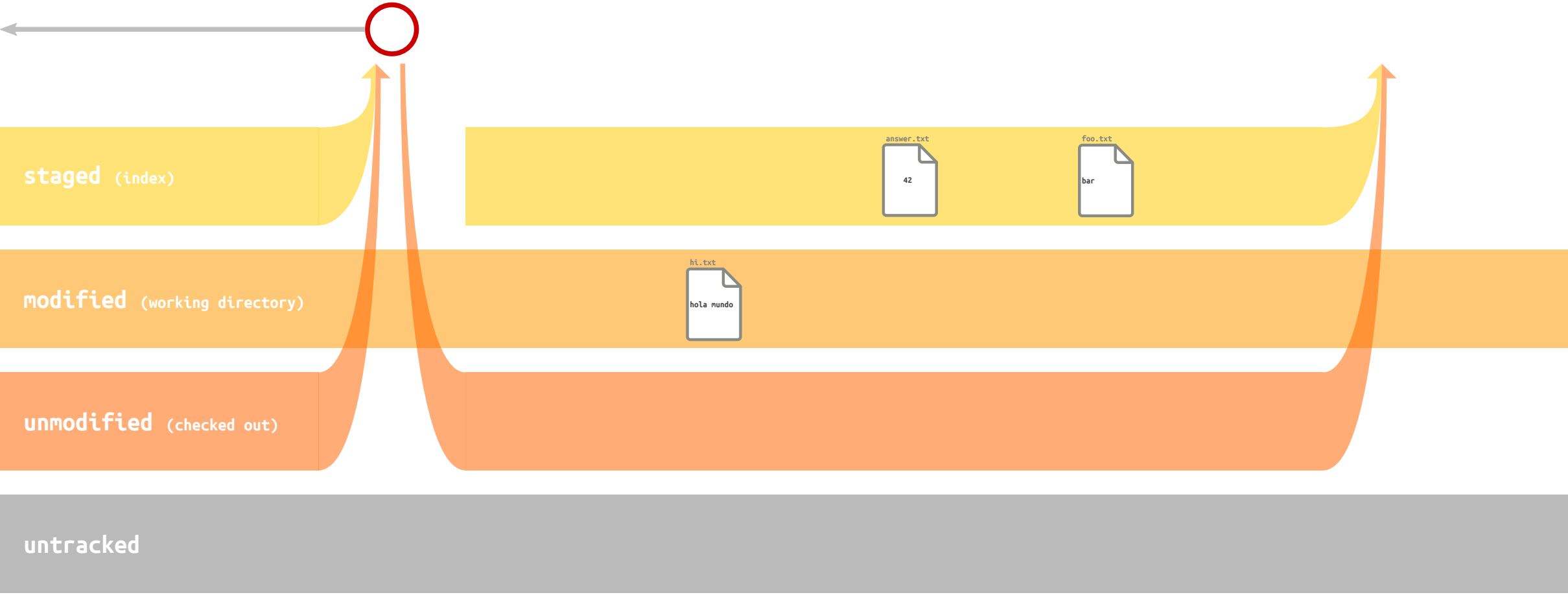
Reverting files to their "in-the-previous-commit" state

CAUTION: you are deleting content, and it's not possible to undo!
New files aren't impacted by git checkout, because they weren't in the previus commit



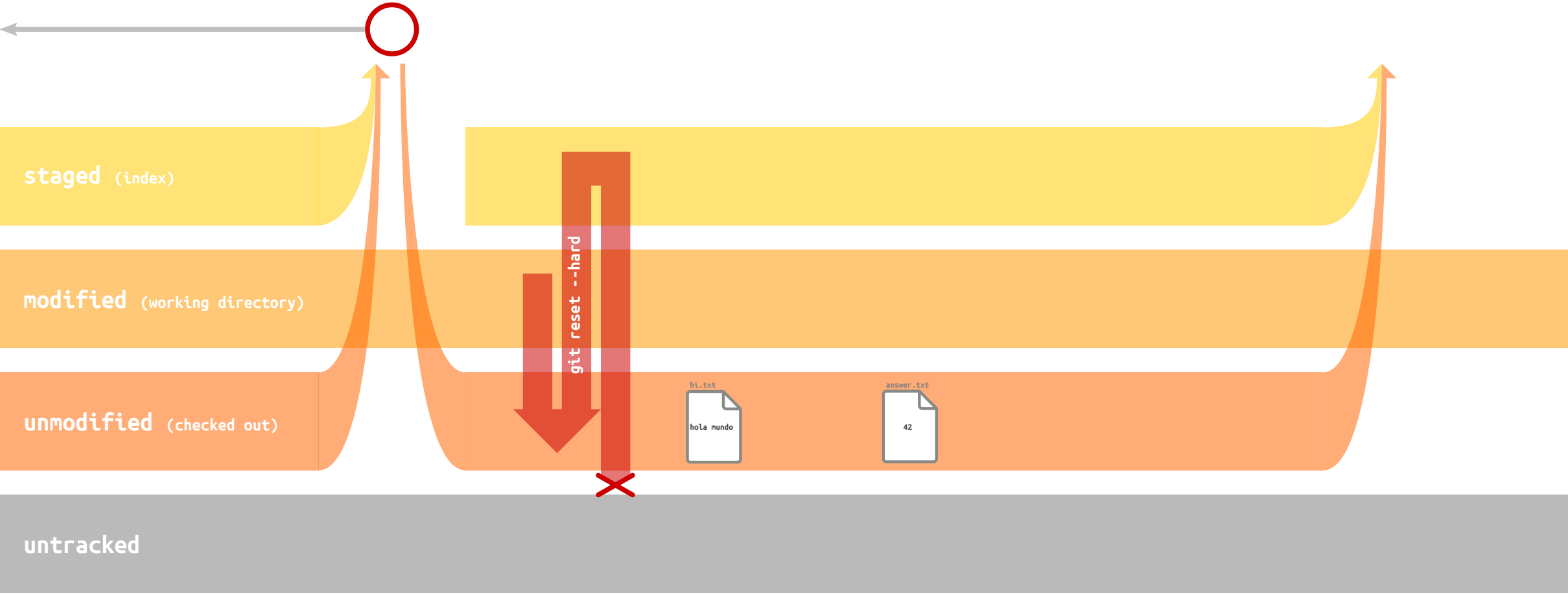

```
git checkout HEAD .    CAUTION!
```

Reverting directory to the "in-the-previous-commit" state



Reverting directory to the "in-the-previous-commit" state

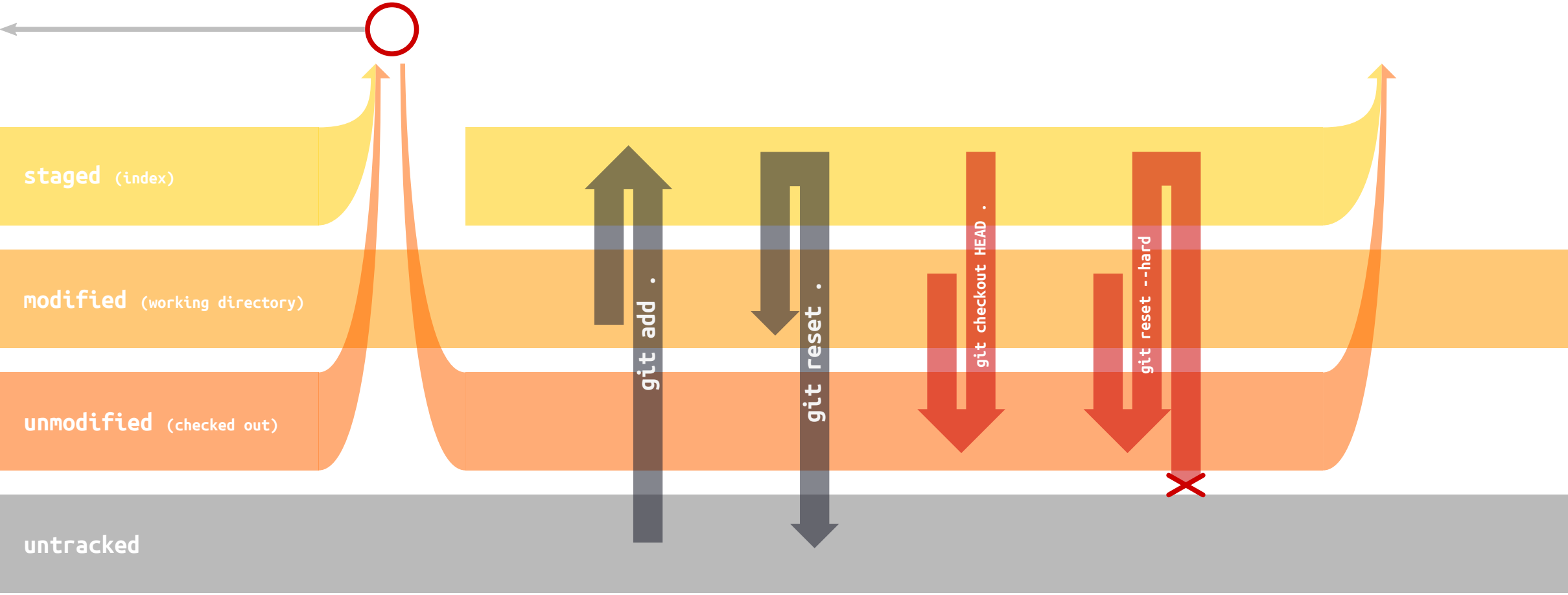
CAUTION: you are deleting content, and it's not possible to undo!
Note that new files are fully discarded (because they didn't exist in previous commit)!




```
git reset --hard    CAUTION!
```

Changing the state of all files of the directory

Let's wrap up



Recap: Undoing things

git reset is the exact undo of git add.

"Unmodifying" a file, using ***git checkout***, **is dangerous** because content will be lost.

git reset --hard **is even more dangerous**, because it deletes all untracked files.

Both *git reset* and *git checkout* can be applied on single files or all of them. *git reset --hard* is on all files only.

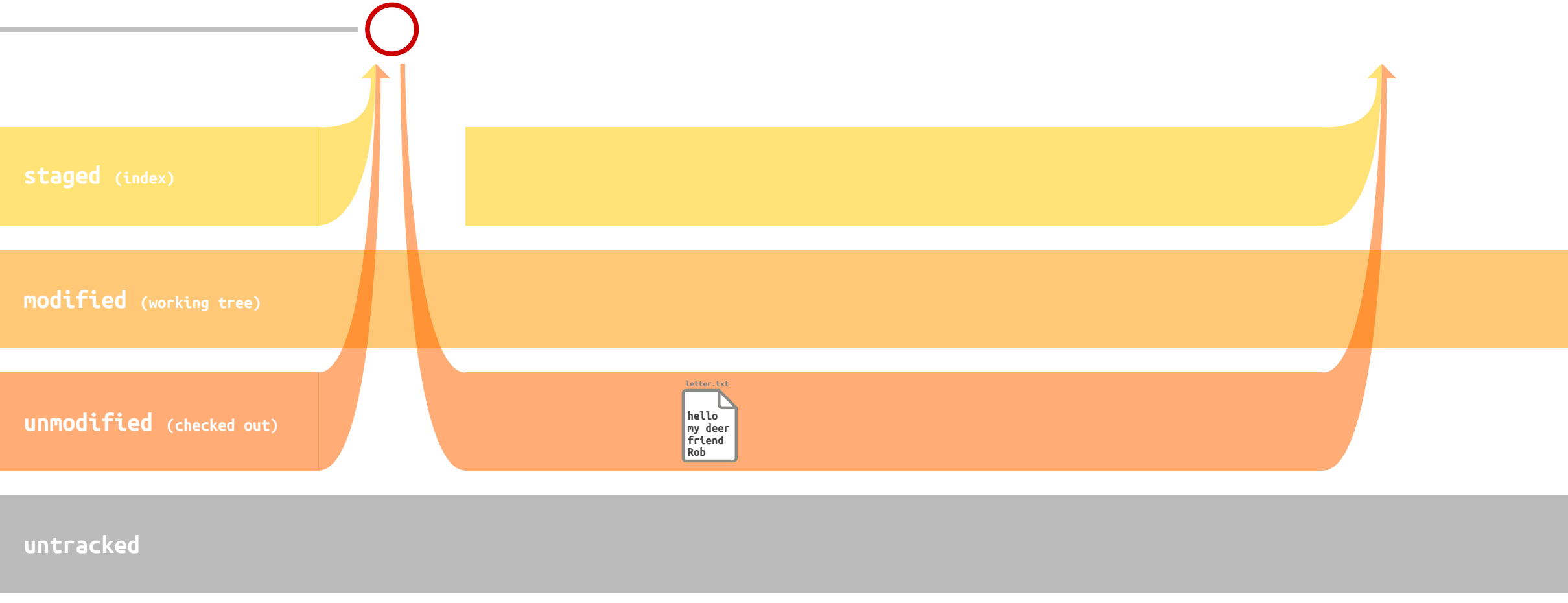
git

What happens at the file-level

- The 5 (well, 4) file states
  - Or is it 2?
- The circle of life (of a file)
  - From one commit to another
- Anatomy of a commit
  - With all the bloody details
- Knowing what is happening (from the command line)
  - Can be usefull… sometimes
- Undoing things
  - Can be usefull… too
- **Going the extra mile**
  - **With more complex yet super cool commands**

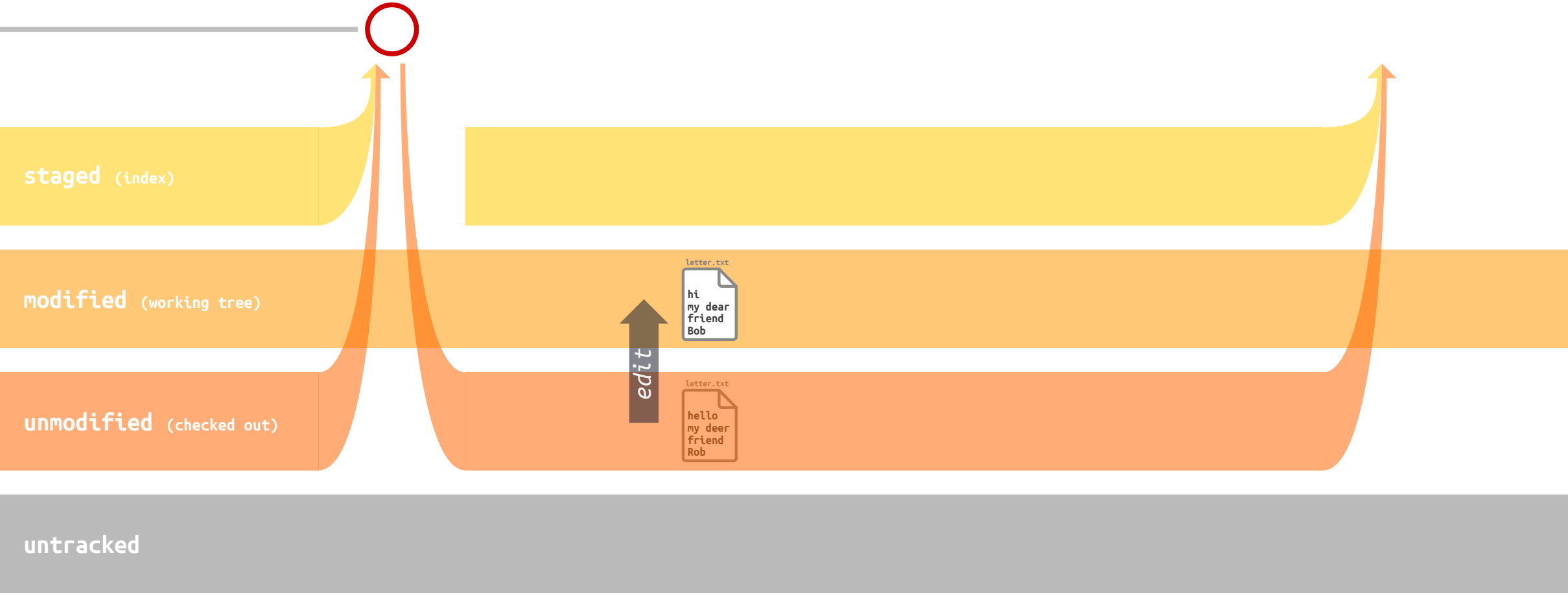
git add --patch

Manage code chunks, and not whole files.



git add --patch

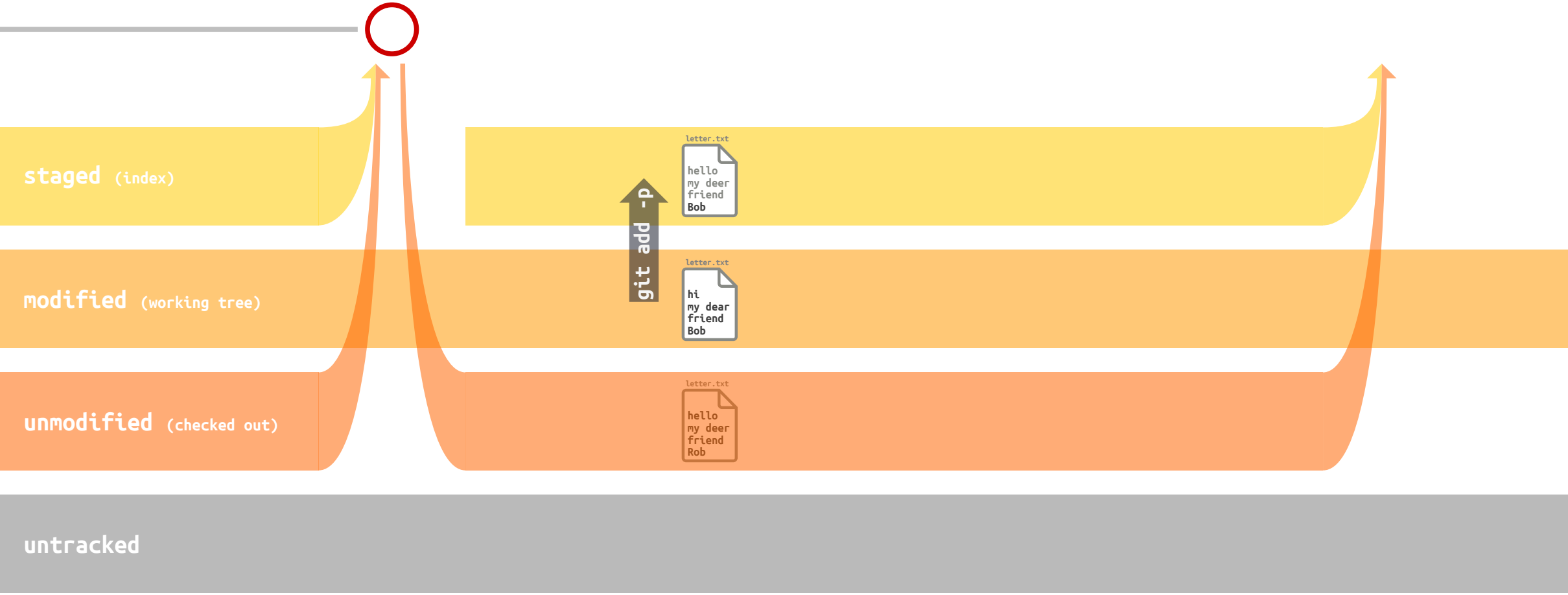
Manage code chunks, and not whole files.




```
echo "hi\nmy dear\nfriend\nBob" > letter.txt
```

git add --patch

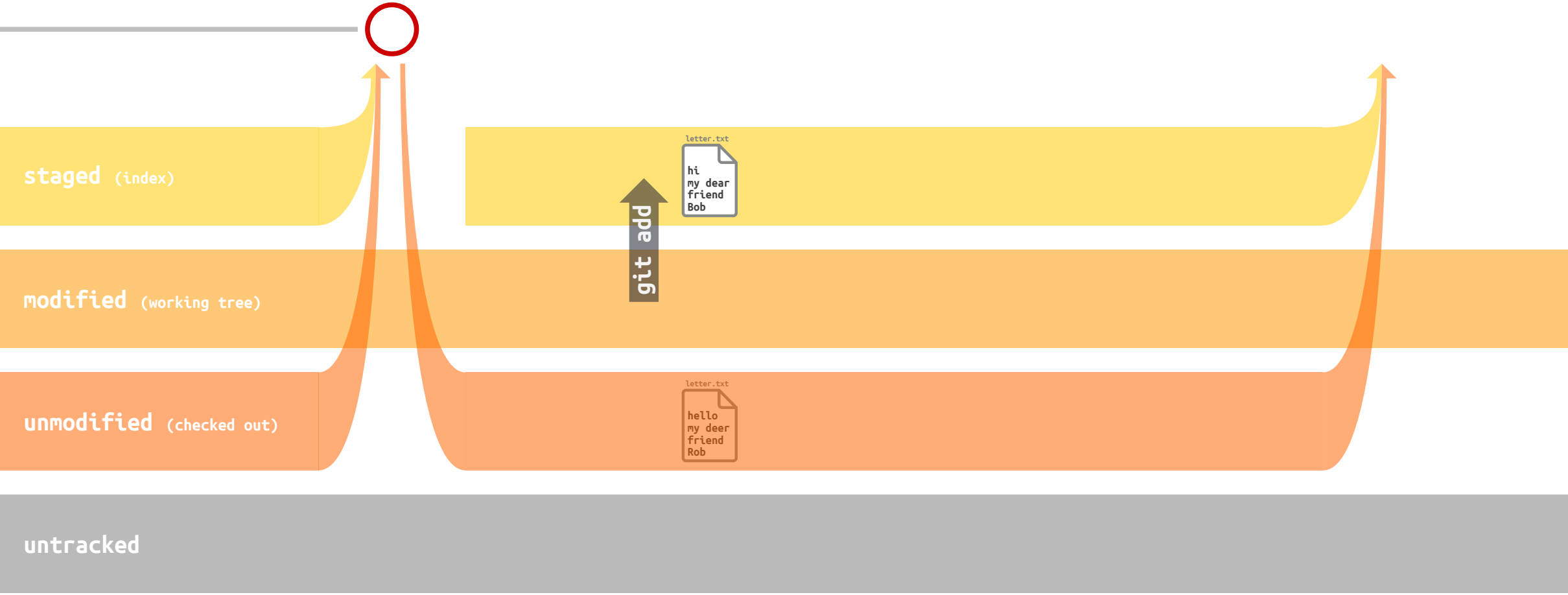
Manage code chunks, and not whole files.




```
git add --patch letter.txt
  […]
```

git add --patch

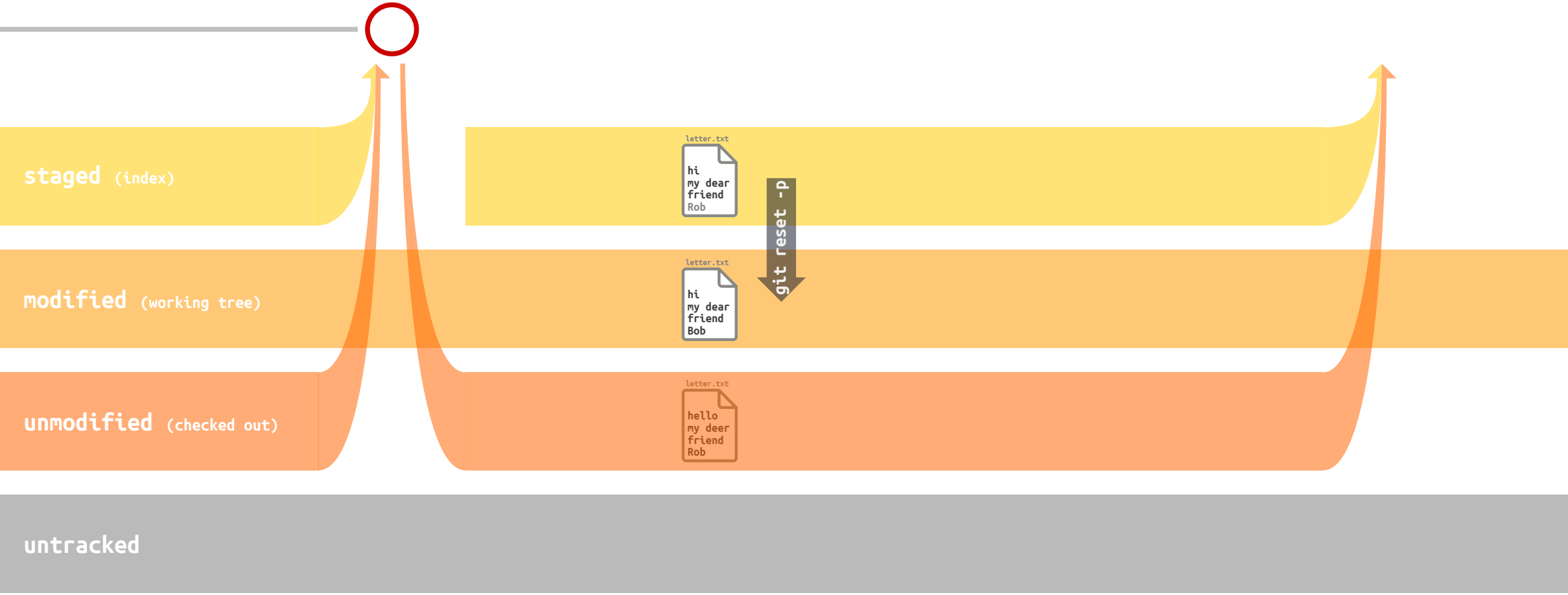
Manage code chunks, and not whole files.




```
git add letter.txt
```

git reset --patch

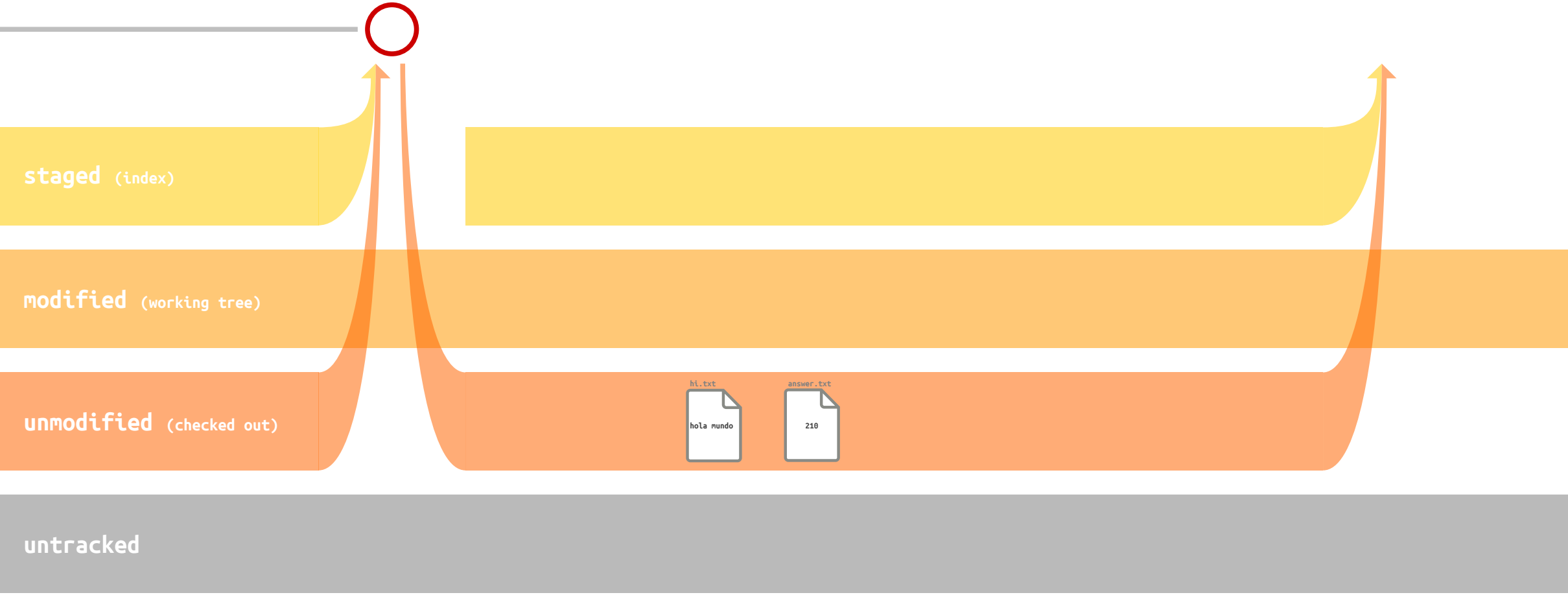
Jurst like git add -p: manage code chunks, and not whole files.




```
git reset -p letter.txt
  […]
```

git stash is an on-the-side back-up of current modifications

You retrieve a "just like after previous commit" state… without loosing your changes!



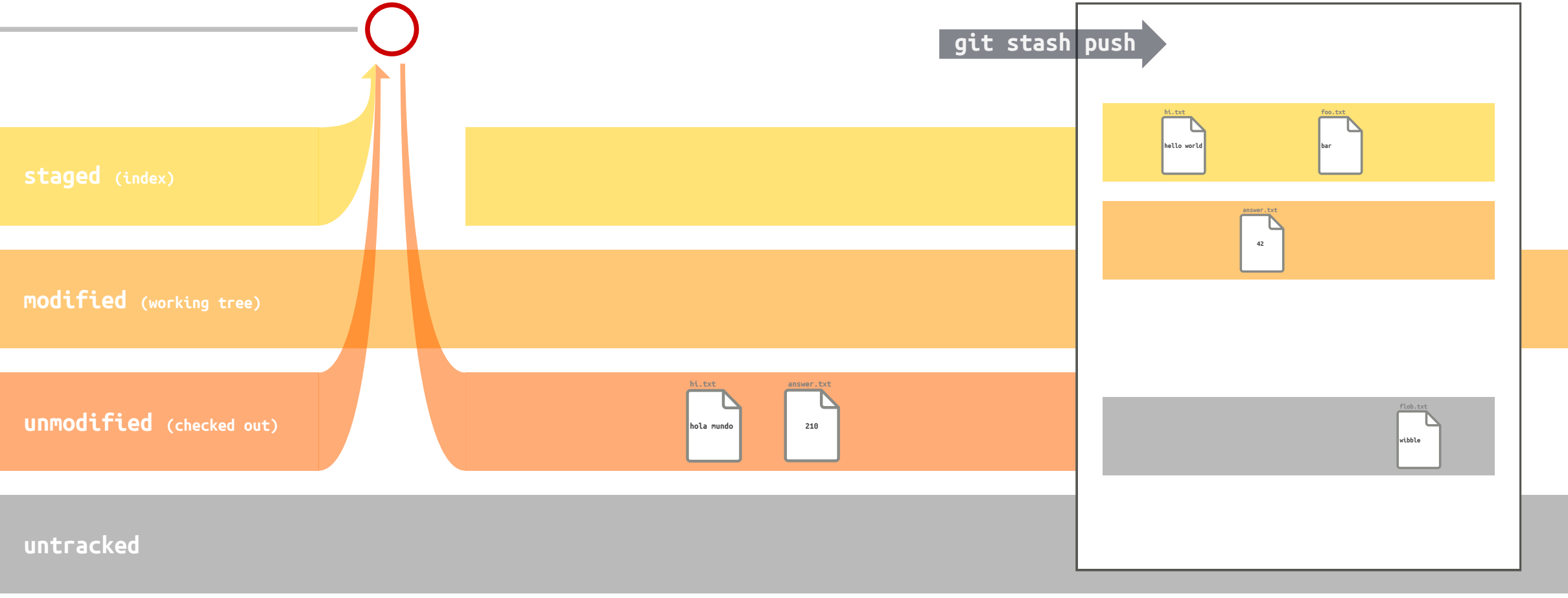
git stash is an on-the-side back-up of current modifications

You retrieve a "just like after previous commit" state… without loosing your changes!


```
echo "42" > answer.txt & echo "hello world" > hi.txt & git add hi.txt
echo "wibble" > flob.txt & echo "bar" > foo.txt & git add foo.txt
```

git stash is an on-the-side back-up of current modifications

You retrieve a "just like after previous commit" state… without loosing your changes!

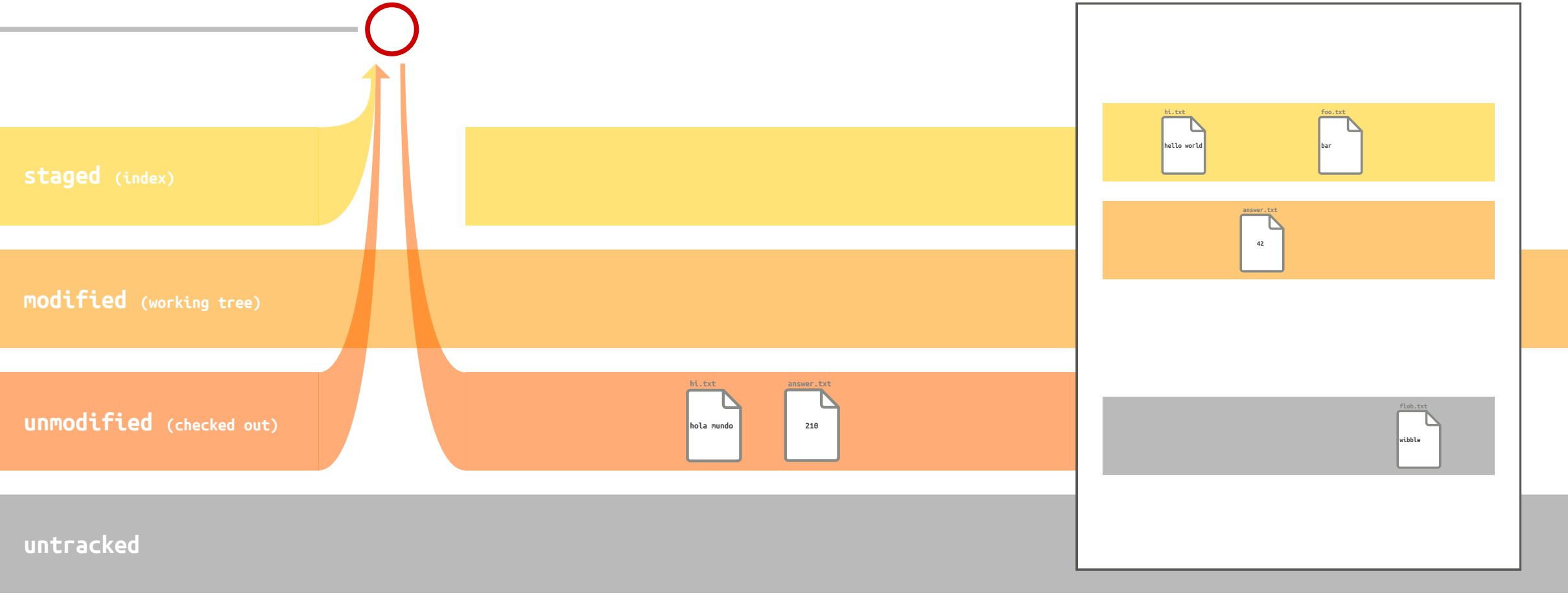



```
git stash push
```

```
git stash
```

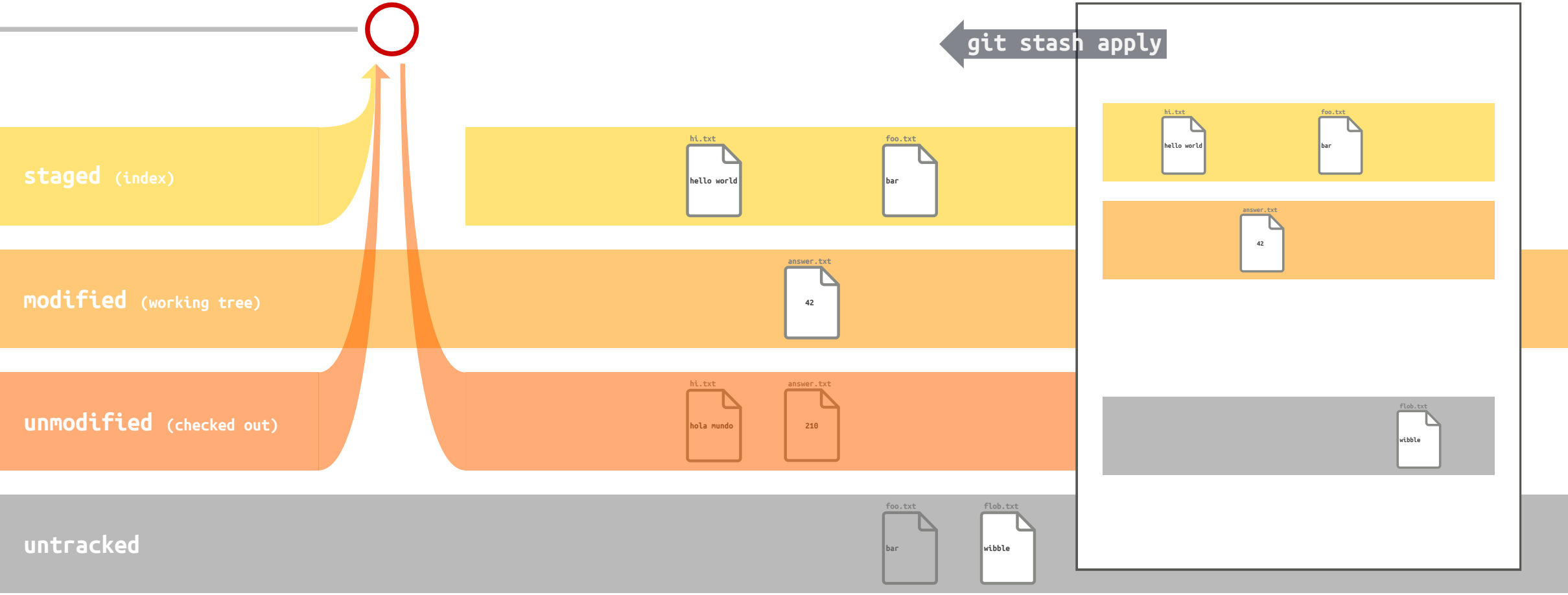
git stash is an on-the-side back-up of current modifications

You can then rebase, checkout another branches, create another commit, …



git stash is an on-the-side back-up of current modifications

You retrieve a "just like after previous commit" state… without loosing your changes!




```
git stash apply
```

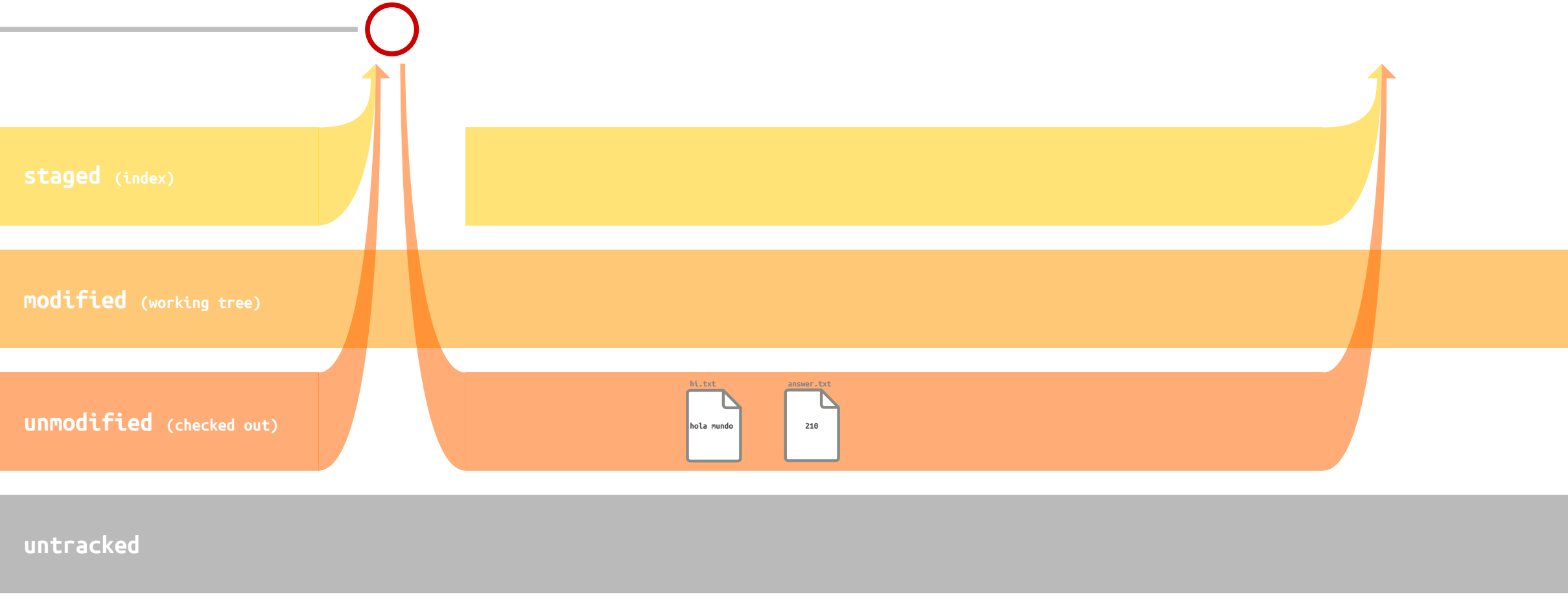
git stash is an on-the-side back-up of current modifications

You retrieve a "just like after previous commit" state… without loosing your changes!


```
git stash apply
git stash drop
```

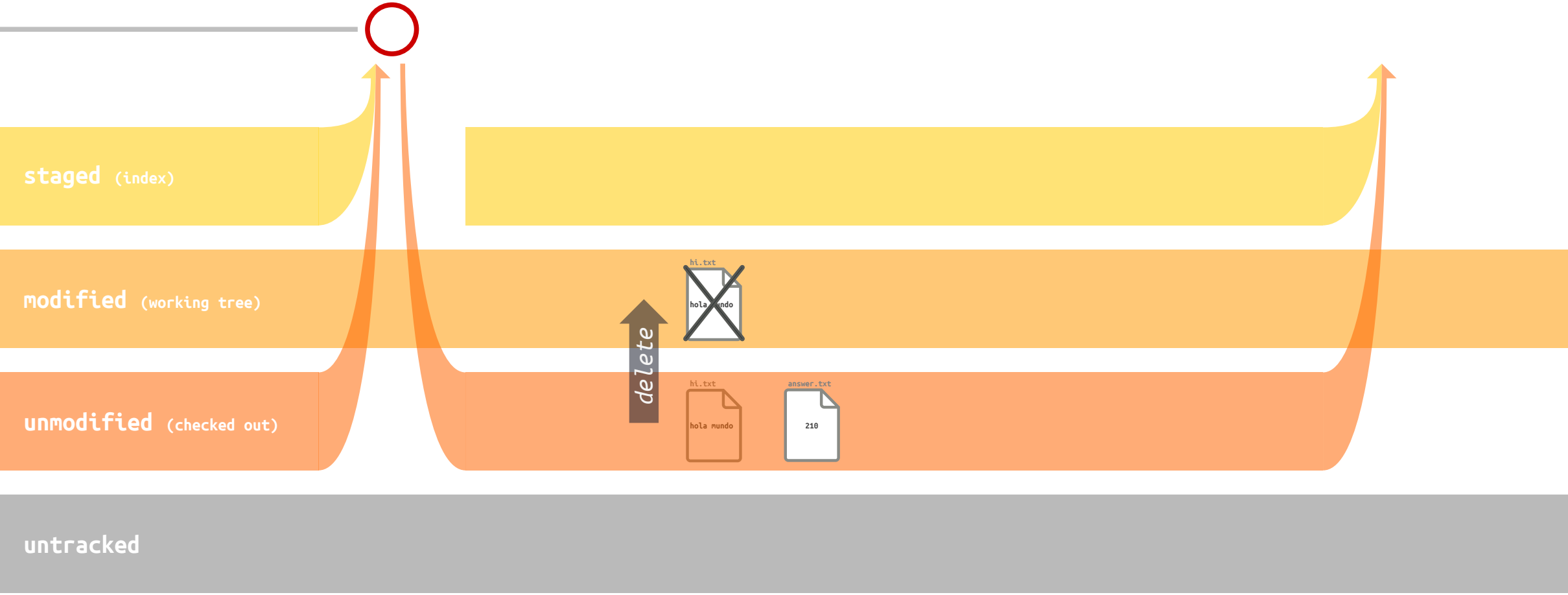
```
git stash pop
```

Deleting files



Deleting files

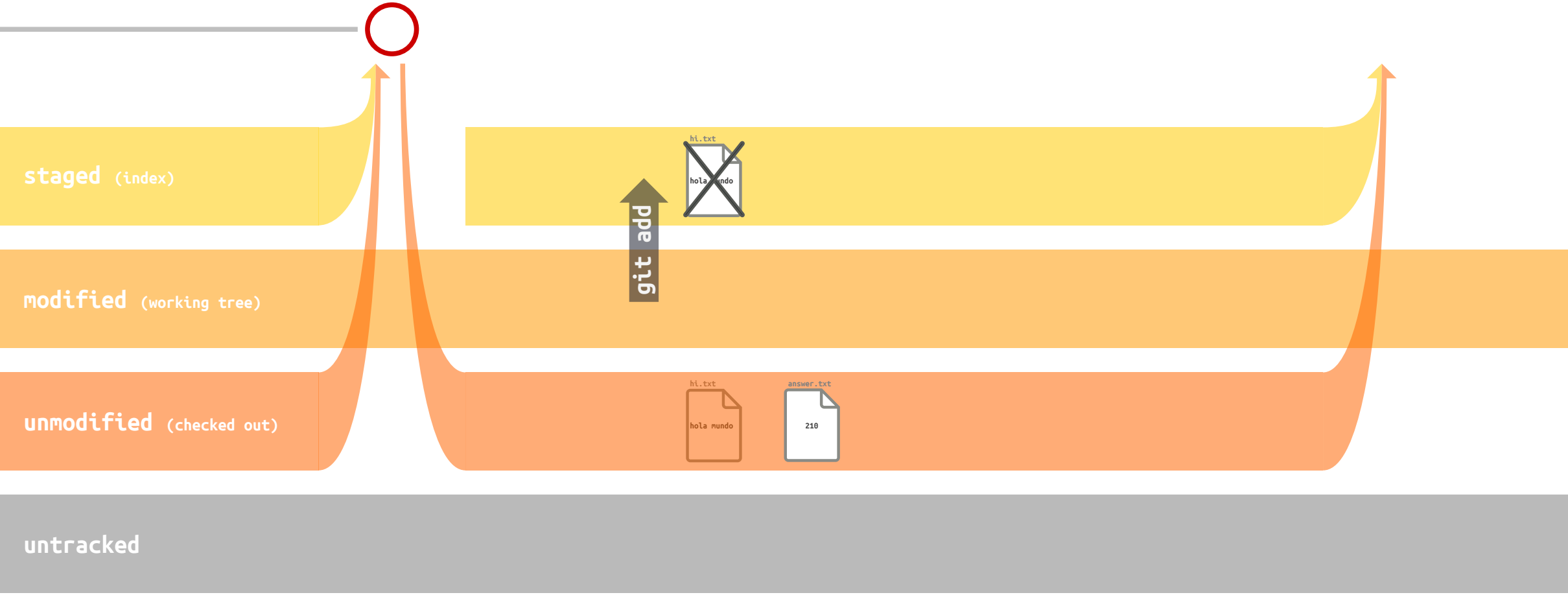
For git, deleting a file is a simply a modification, just like editing it would be.




```
rm hi.txt
```

Deleting files

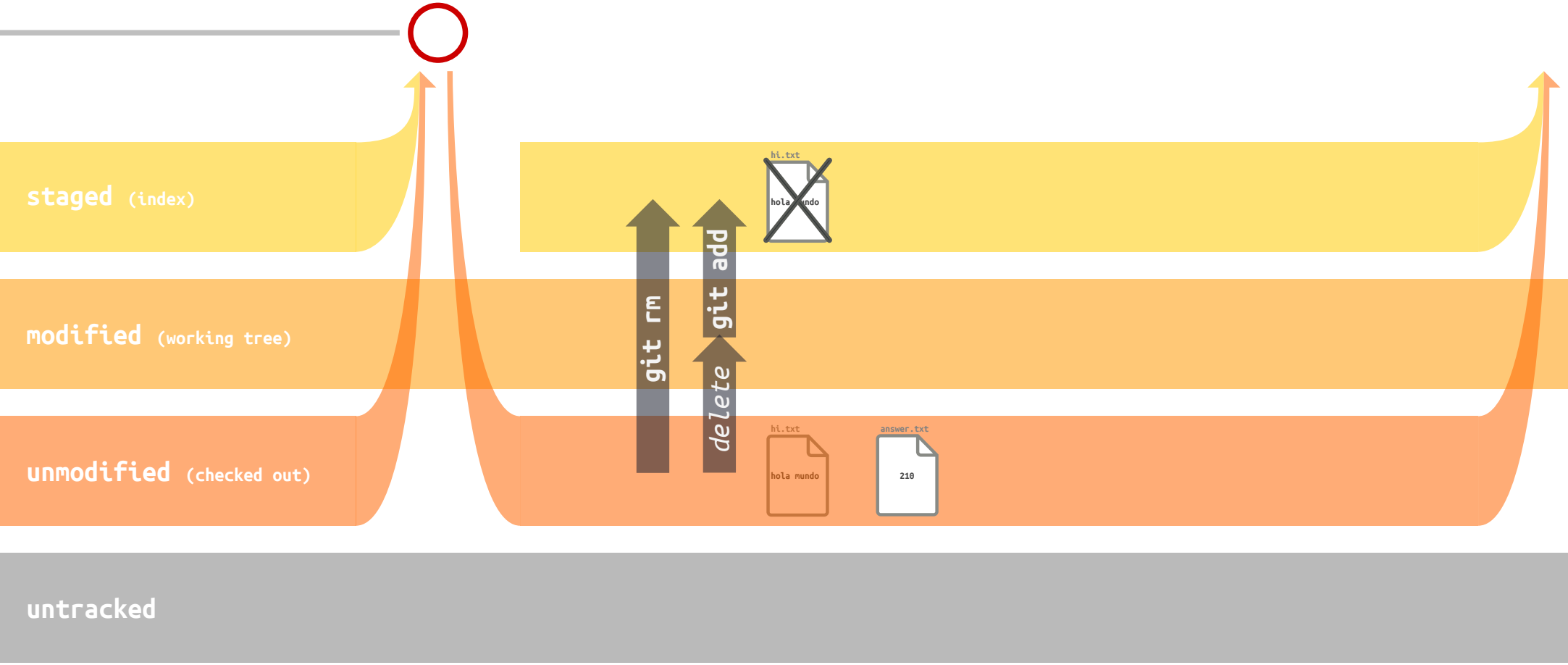
Thus, the (deleted) file must be staged for the next commit to record the deletion.




```
git add hi.txt
```

Deleting files

git rm enables you to perform both operations in one go

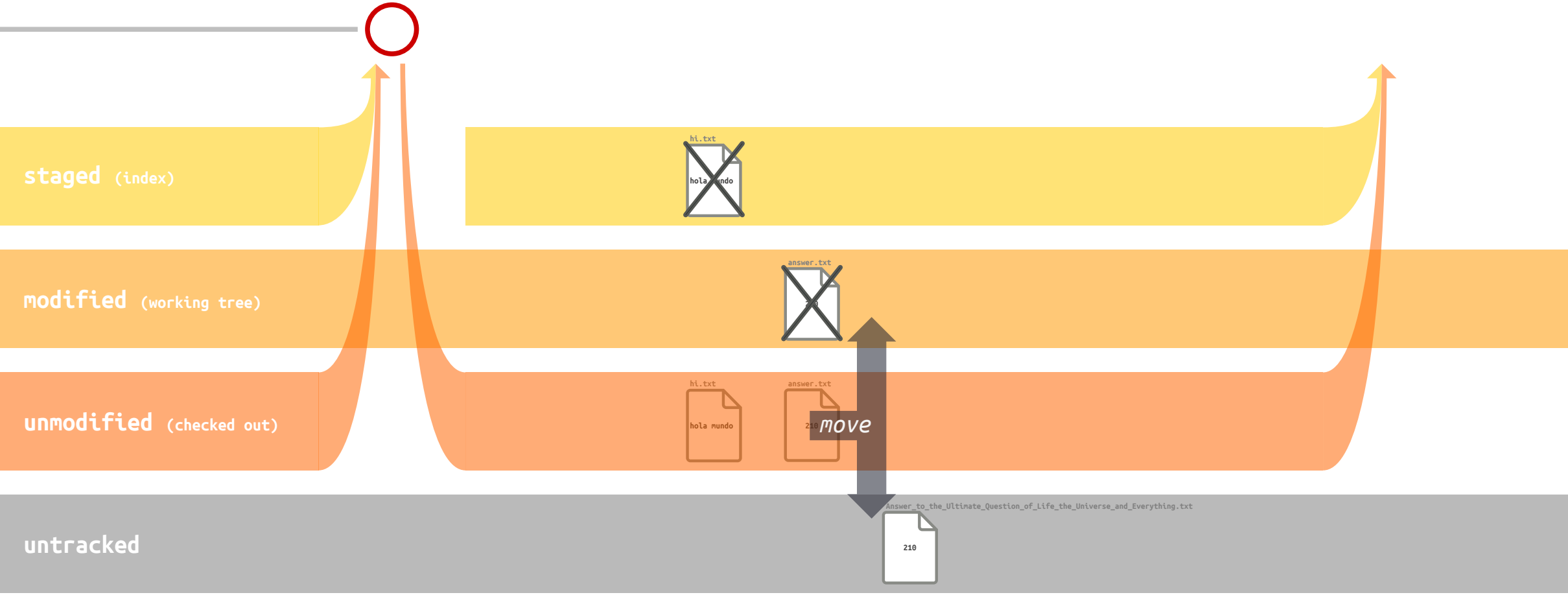



```
rm hi.txt
git add hi.txt
```

```
git rm hi.txt
```

Moving (or renaming) files

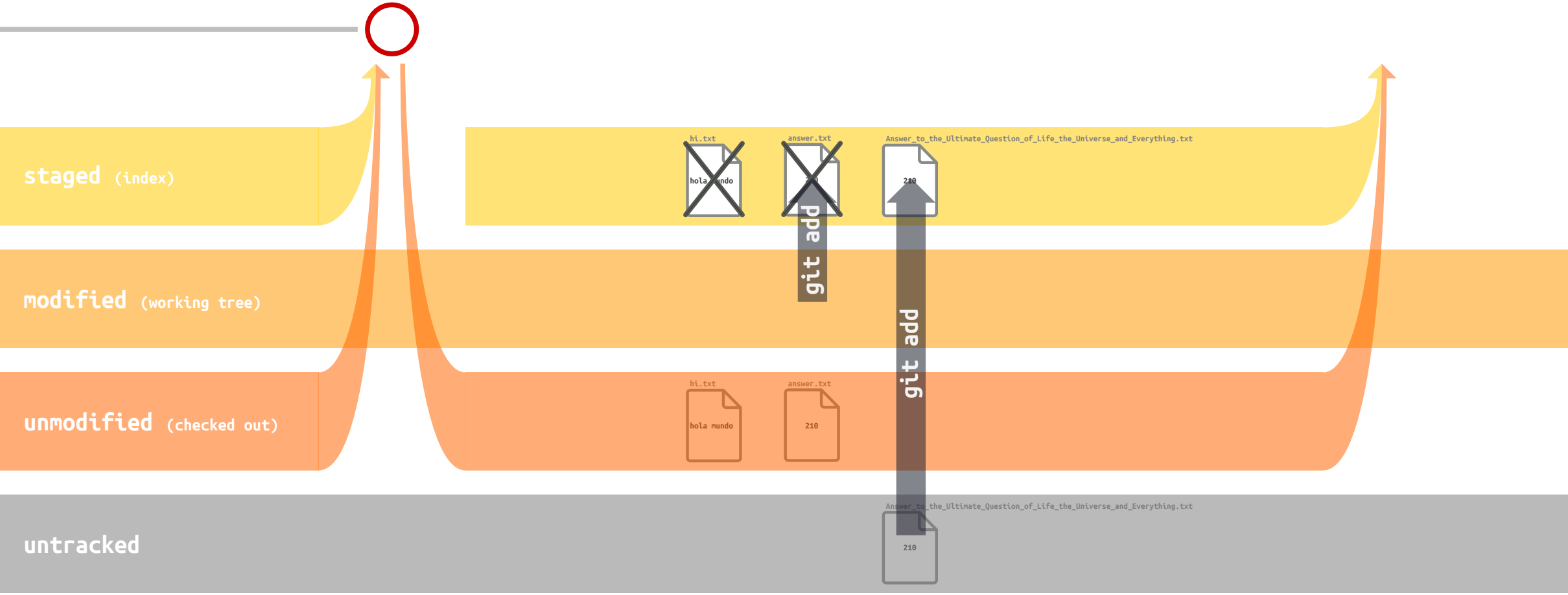
Similarly, for git, moving a file is like deleting the old and then creating a new file.




```
mv answer.txt Answer_to_the_Ultimate_Question_of_Life_the_Universe_and_Everything.txt
```

Moving (or renaming) files

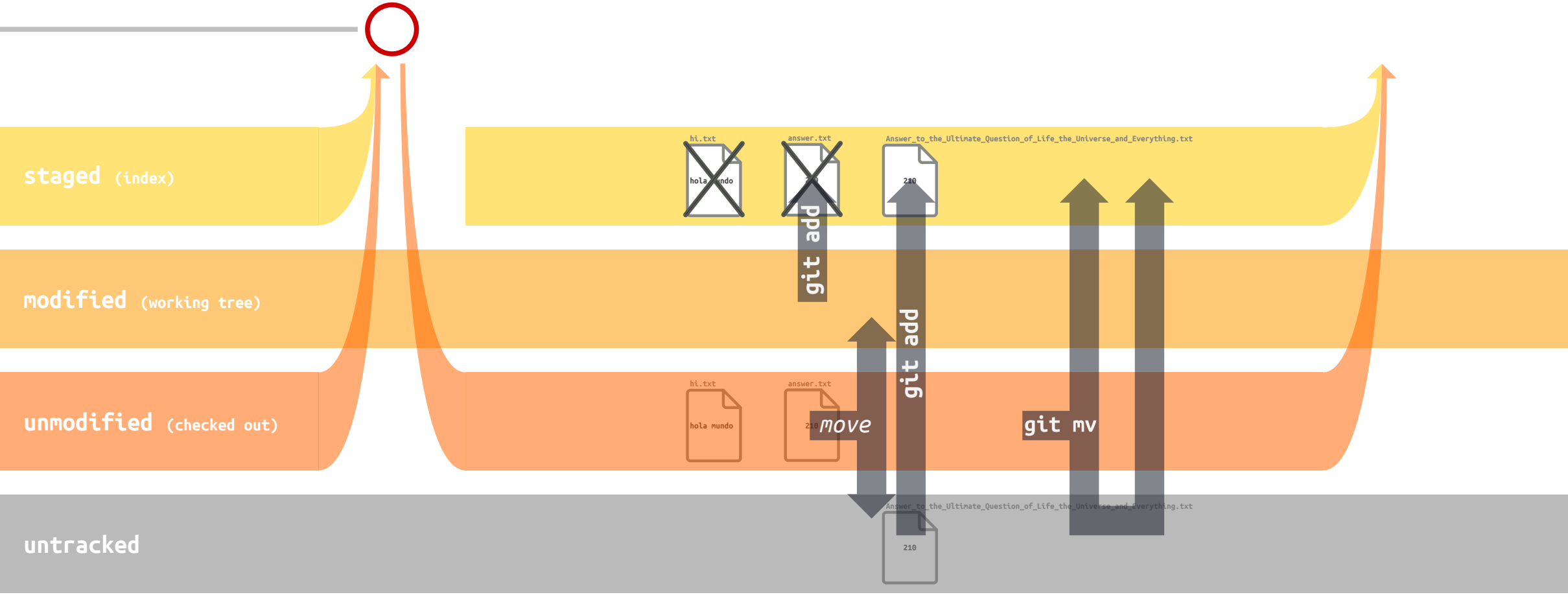
And, similarly again, the old and the new file must both be added.




```
git add answer.txt Answer_to_the_Ultimate_Question_of_Life_the_Universe_and_Everything.txt
```

Moving (or renaming) files

git mv enables you to perform all the 3 operations in one go

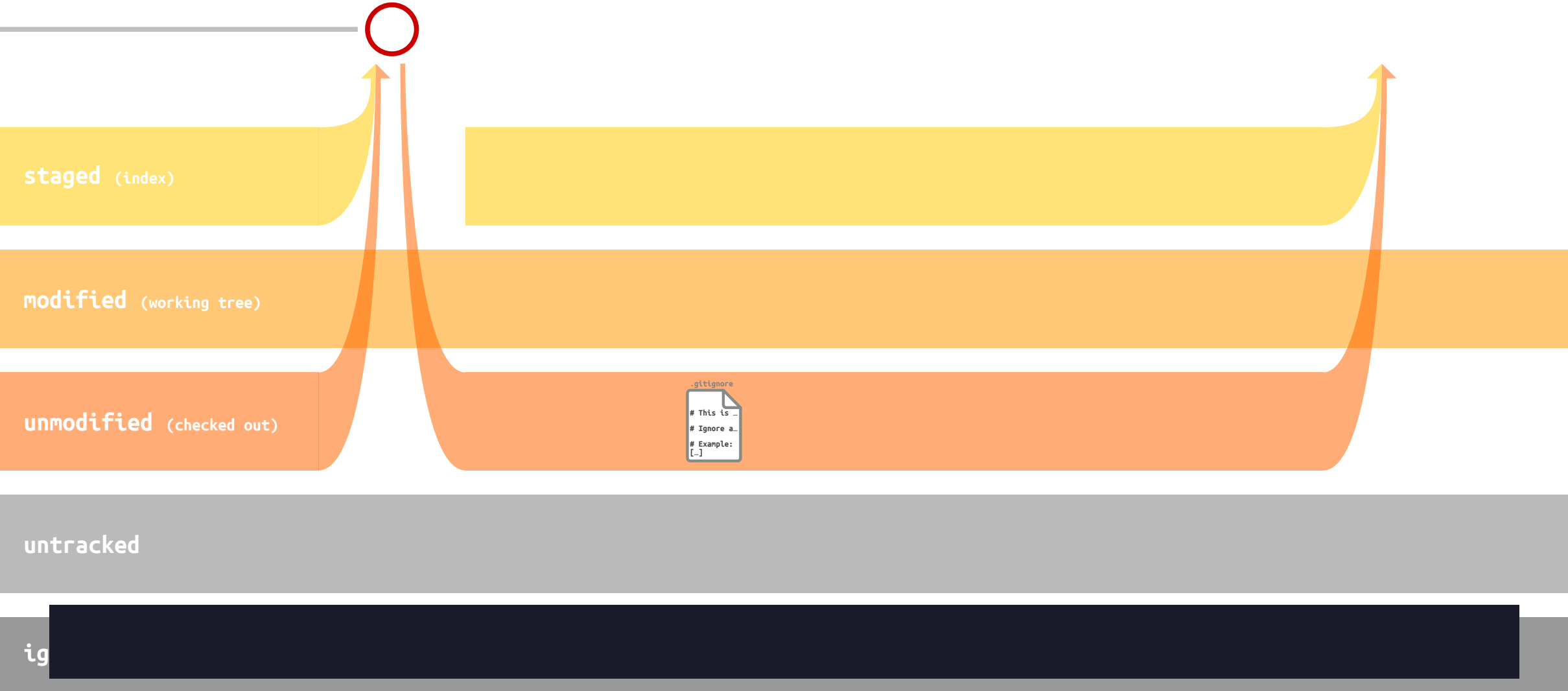



```
mv answer.txt Answer_to_the_Ultimate_Question_of_L
git add answer.txt Answer_to_the_Ultimate_Question
```

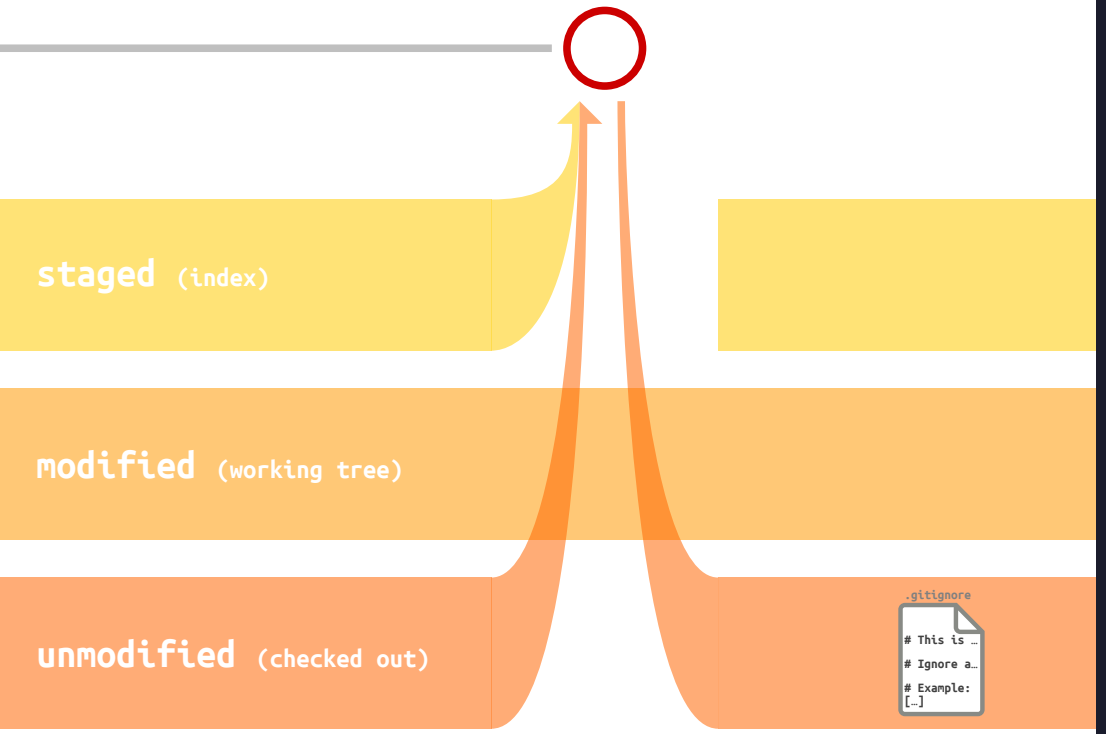
```
git mv answer.txt Answer_to_the_Ultimate_Question_
```

The **.gitignore** file tells git which files to ignore

The file must be located at directory's root



.gitignore is the way to tell git which files to ignore




```
cat .gitignore
# This is a comment.

# Ignore any files whose name begins with `hello`, in all directories.
# Example:
#    hello.txt
#    hello.c
#    dir/hello.log
hello.*

# Make an exception to rules above for (i.e. do not ignore) `hello.jpg` files.
# Example:
#    hello.jpg
#    dir/hello.jpg
!hello.jpg

# Ignore any file whose name begins with `bye` – at directory's root only.
# Example:
#    bye.txt
#    bye.c
#    BUT NOT: dir/bye.log
/bye.*

# Ignore any file in the `logs` directory, if the latter is at directory's root.
# Example:
#    logs/foo.log
#    BUT NOT dir/logs/bar.log
logs/**

# Ignore any file in any `temp` directory.
# Example:
#    temp/foo.txt
#    dir/temp/bar.txt
**/temp/**
```

The **.gitignore** file tells git which files to ignore

Note that how git does not track directories

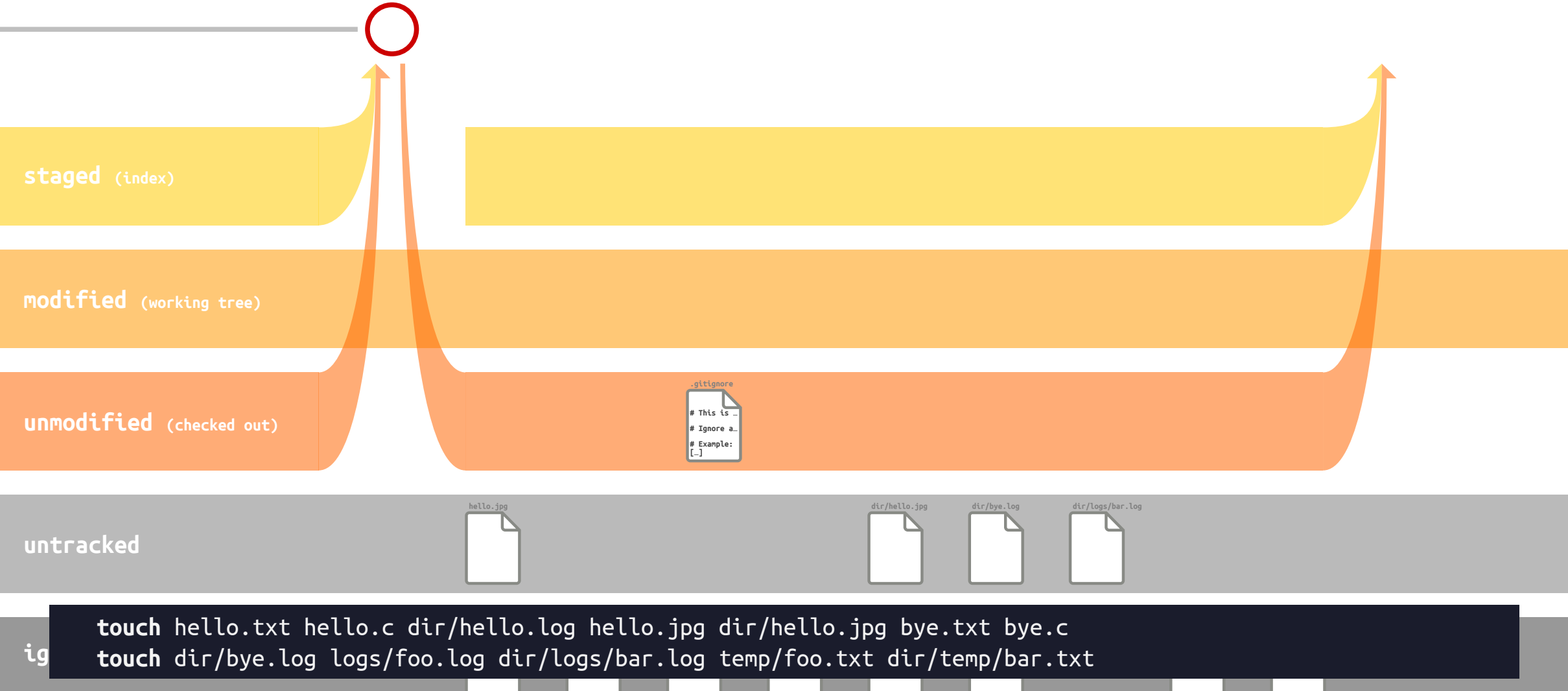
untracked

ig

```
mkdir dir logs dir/logs temp dir/temp
```

The **.gitignore** file tells git which files to ignore

For git, files in directories are just like files at root with `/` in their name

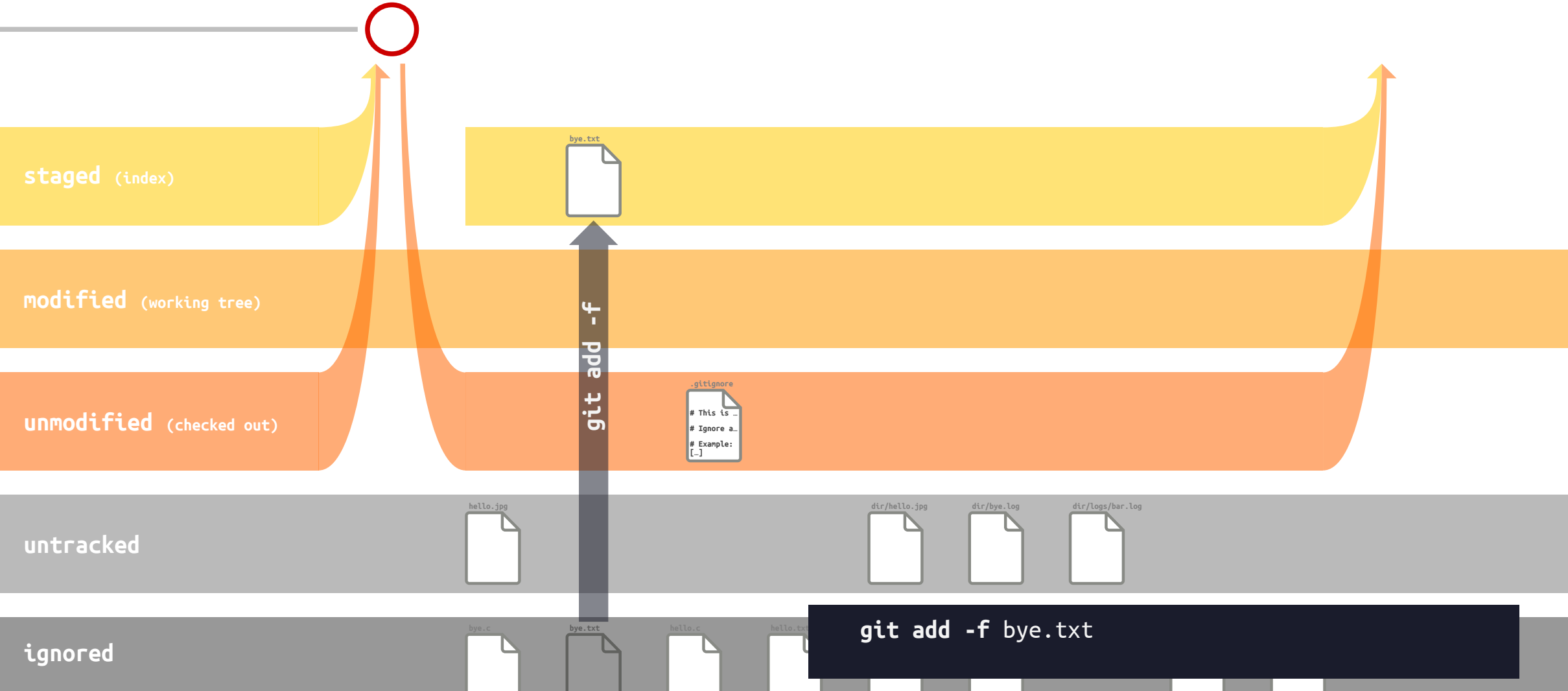



```
touch hello.txt hello.c dir/hello.log hello.jpg dir/hello.jpg bye.txt bye.c
touch dir/bye.log logs/foo.log dir/logs/bar.log temp/foo.txt dir/temp/bar.txt
```

The **.gitignore** file tells git which files to ignore

git add --force surpasses .gitignore

Similarly, a tracked file (i.e. present in previous commit) supersede .gitignore rules



Recap: the extra 1.60934 km

The ***--patch*** (or ***-p***) flag enables to **manipulate code chuncks, and not whole files**. This is notably useful with ***git add*** and ***git reset***.

git stash takes current modifications and **stores them on the side**. You can then notably change the checked out commit.

After deleting, moving, or renaming a file, its **old path must be staged** (***git add***) for git to record the change. For moving and renaming, the new file need also to be staged. ***git rm*** and ***git mv*** do both operation (on disk and ***git add***) in one go.

The **.gitignore** file (at directory's root) tells git which files to ignore. It can be overridden using ***git add -f***.

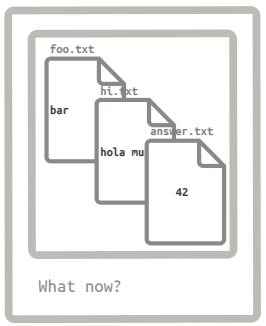
git

one more thing

A commit is more than the snapshot

but what is a files snapshot?

9c435a86e664be00db0d973e981425e4a3ef3f8d

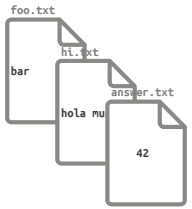


Étienne Boisseau-Sierra <ebosi@unipart.io>
Wed Oct 16 15:45:42 2019 +0100
Étienne Boisseau-Sierra <ebosi@unipart.io>
Wed Oct 16 15:45:42 2019 +0100

=

unique ID 9c435a86e664be00d

files snapshot



message What now?

ID of the parent(s) 0d973e9c4353ef3f8ddb98a8

author Étienne Boisseau-Sierra <ebosi@unipart.io>

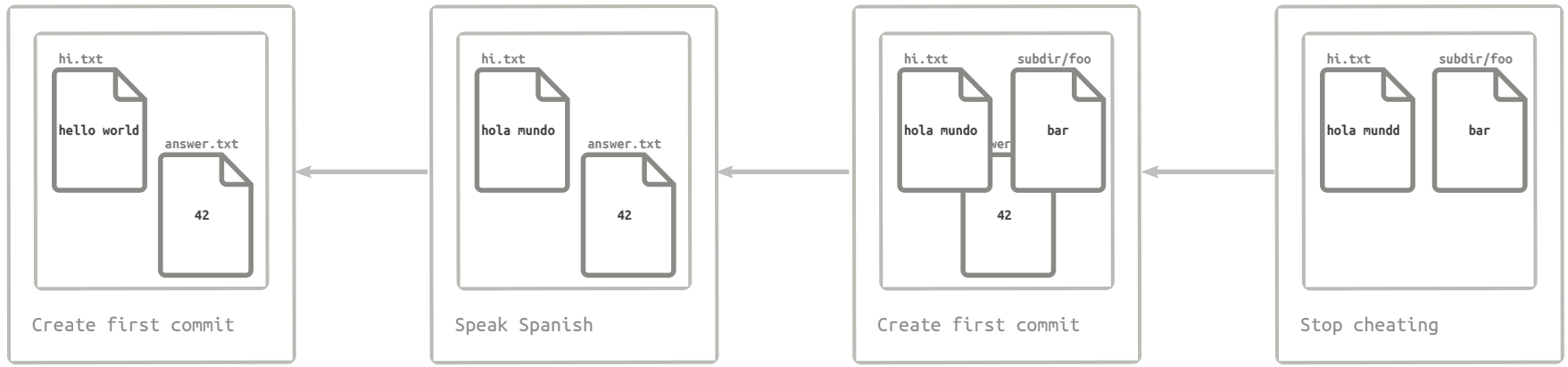
date Wed Oct 16 15:45:42 2019 +0100

committer Étienne Boisseau-Sierra <e

committer date Wed Oct

A files snapshot is a snapshot

not a delta



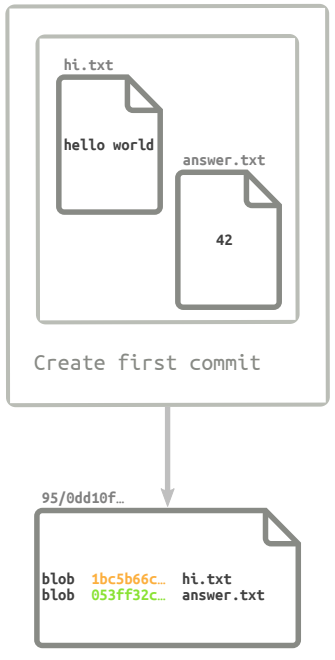
hi.txt

answer.txt

subdir/foo

A files snapshot is a tree

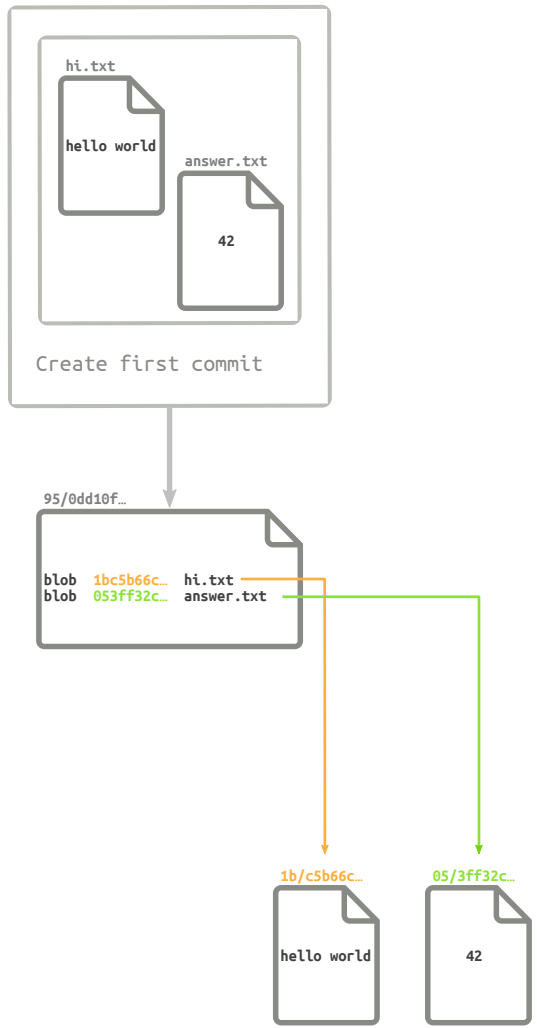
which points to files' checksums



.git/objects/

A files snapshot is a tree

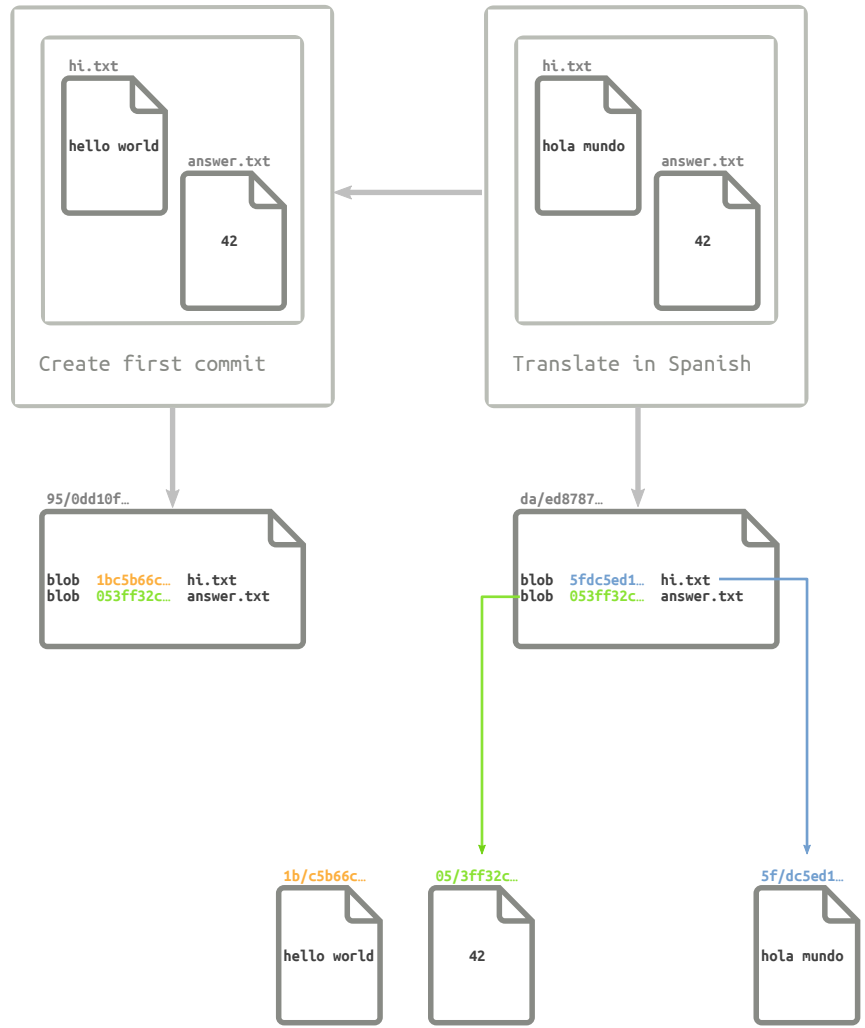
which points to files' checksums (which locate their content)



.git/objects/

A files snapshot is a tree

which points to files' checksums (which locate their content)

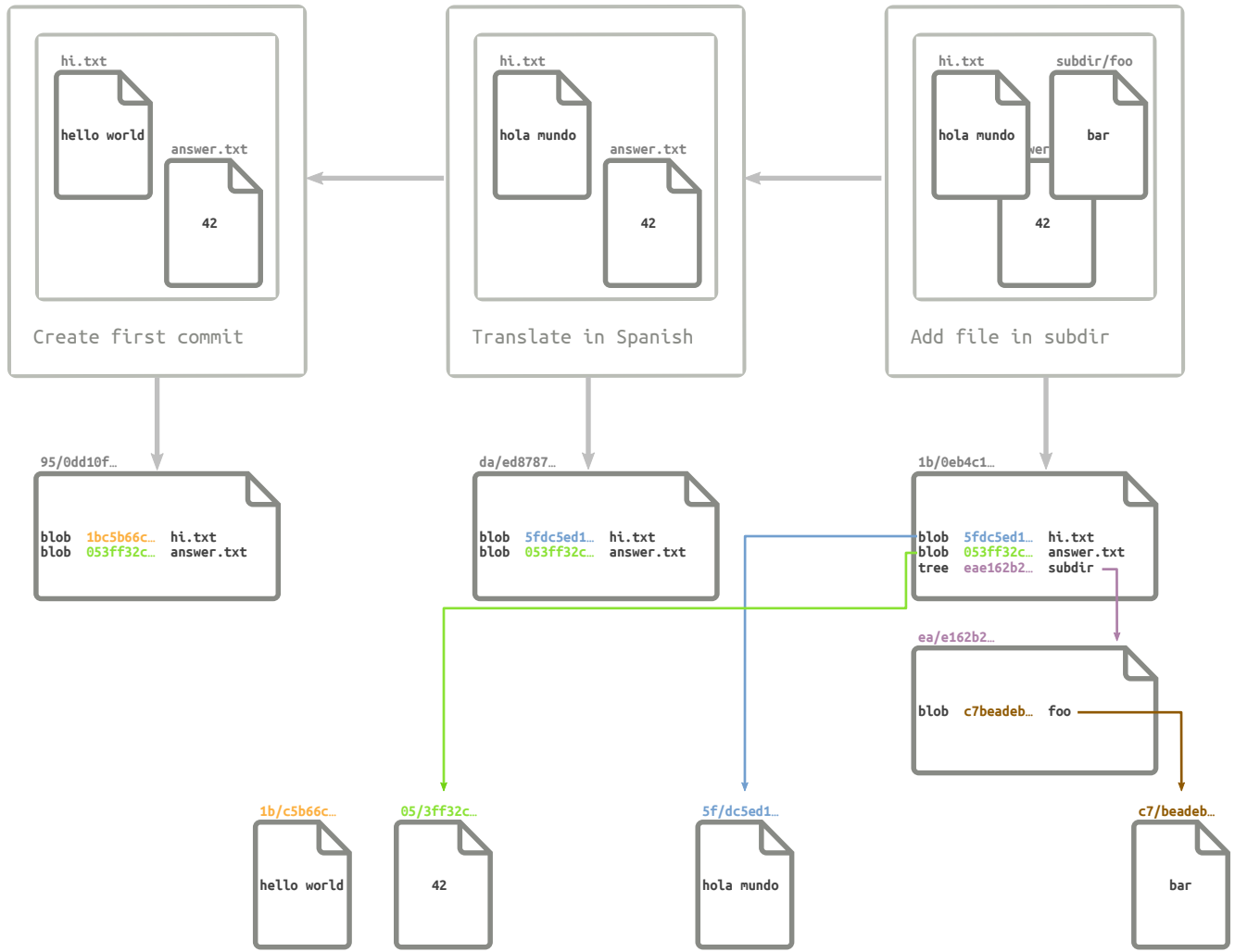


.git/objects/

A files snapshot is a tree

which points to files' or trees' checksums (which locate their content)

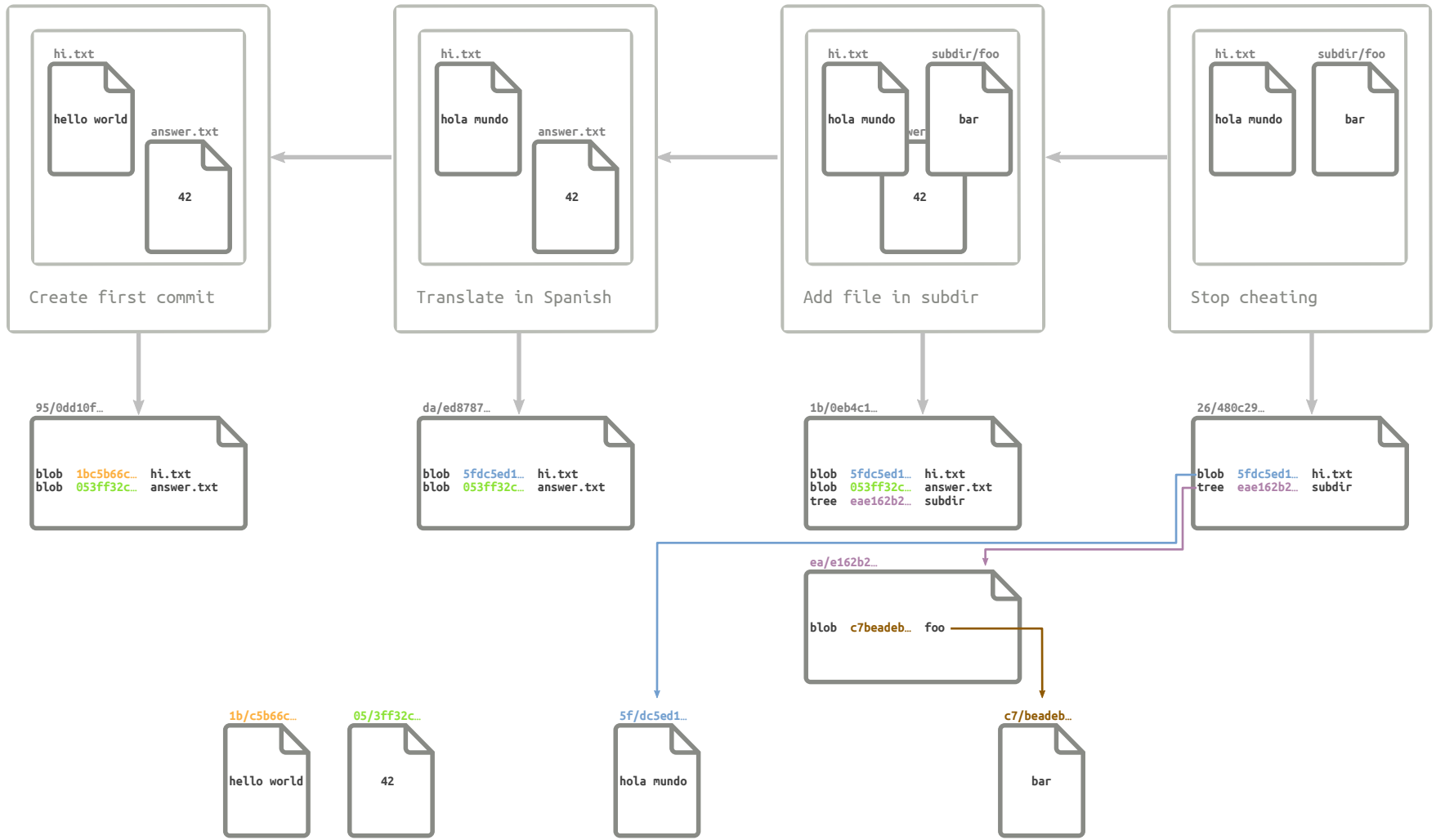
.git/objects/



A files snapshot is a tree

which points to files' or trees' checksums (which locate their content)

.git/objects/



A commit is an object

just like files and trees

=

```
ls .git/objects/9c | grep 435a
435a86e664be00db0d973e981425e4a3ef3f8d
```

A commit is an object

just like files and trees

=

```
ls .git/objects/9c | grep 435a
435a86e664be00db0d973e981425e4a3ef3f8d


git cat-file -p 9c435a86e
tree 1b0eb4c1…
parent 0c973e9c4353ef3f8ddb98a8…
author Étienne Boisseau-Sierra <ebosi
    @unipart.io> 1571223942 +0100
committer Étienne Boisseau-Sierra <ebosi
    @unipart.io> 1571223942 +0100

What now?
```