Git concepts for engineers

Table of Contents

[Introduction 3](#_Toc378943135)

[Scope of this document 3](#_Toc378943136)

[Overview 3](#_Toc378943137)

[Miscellaneous Basic Concepts 5](#_Toc378943138)

[Git tracks content 5](#_Toc378943139)

[The working tree 6](#_Toc378943140)

[Nearly every operation is local and fast 7](#_Toc378943141)

[Miscellaneous small items 7](#_Toc378943142)

[SHA1 7](#_Toc378943143)

[References 8](#_Toc378943144)

[Git objects / object database 8](#_Toc378943145)

[Commits 8](#_Toc378943146)

[Commit id is a sha1 8](#_Toc378943147)

[Commit history is a DAG 9](#_Toc378943148)

[Commit (action) 10](#_Toc378943149)

[Commit (revision) 10](#_Toc378943150)

[Checkout, switch 10](#_Toc378943151)

[Commit properties 10](#_Toc378943152)

[The HEAD commit 10](#_Toc378943153)

[Index aka staging area 10](#_Toc378943154)

[Tags 13](#_Toc378943155)

[Branches 13](#_Toc378943156)

[What is a branch? 13](#_Toc378943157)

[The master branch 14](#_Toc378943158)

[The current branch 14](#_Toc378943159)

[Deleting branches or commits 15](#_Toc378943160)

[Pack files 16](#_Toc378943161)

[Merging 16](#_Toc378943162)

[Automatic merge with no merge conflicts 16](#_Toc378943163)

[Merge conflicts 17](#_Toc378943164)

[Fast forward merge 17](#_Toc378943165)

[Squash merge 18](#_Toc378943166)

[Merge concepts 18](#_Toc378943167)

[Rebase 19](#_Toc378943168)

[Changing History / Fixing Mistakes 19](#_Toc378943169)

[Amend a commit 20](#_Toc378943170)

[Distributed Git 20](#_Toc378943171)

[Distributed workflows 21](#_Toc378943172)

[Clone 21](#_Toc378943173)

[Remote tracking branch 21](#_Toc378943174)

[Upstream (aka tracking) branch 22](#_Toc378943175)

[Fetch 22](#_Toc378943176)

[Pull 23](#_Toc378943177)

[Push 24](#_Toc378943178)

[Miscellaneous Advanced Concepts 26](#_Toc378943179)

[The stash 26](#_Toc378943180)

[Git has integrity / trust 26](#_Toc378943181)

[Blame (aka annotate or praise) 26](#_Toc378943182)

[Bisect 26](#_Toc378943183)

[Pickaxe 27](#_Toc378943184)

[The full DAG 27](#_Toc378943185)

[Git as UML class diagram 28](#_Toc378943186)

[Garbage collector 29](#_Toc378943187)

[Pack files 29](#_Toc378943188)

[Reflog 29](#_Toc378943189)

[No locking of files 29](#_Toc378943190)

[No explicit file moving / renaming 29](#_Toc378943191)

[Attributes 29](#_Toc378943192)

[Grafts 29](#_Toc378943193)

[Replacing Objects 29](#_Toc378943194)

[Patch-Id 29](#_Toc378943195)

[Hooks 29](#_Toc378943196)

[Appendix 30](#_Toc378943197)

[Glossary 30](#_Toc378943198)

[Index 30](#_Toc378943199)

[Bibliography 30](#_Toc378943200)

# Introduction

## Scope of this document

This manual tries to complement existing Git manuals by using a rather technical approach to explain Git. That stems from the author's experience who feels that Git is much easier to use and understand when the technical concepts are properly understood. For an average engineer those technical concepts are rather easy to understand anyway. Linus Torvalds is very right by saying “the data structures are really really really simple”. Most Git documentation is apparently written for a very average user with no technical background, and oversimplifies many concepts, disobeying Albert Einstein’s advice “Make it as simple as possible, but not simpler”. Such oversimplified documentation makes it more complicated at the end in daily life, because you as a user never truly understand what the Git commands really do.

(Chacon) has become much better in the recent, making this document here a bit less important than it was in 2012.

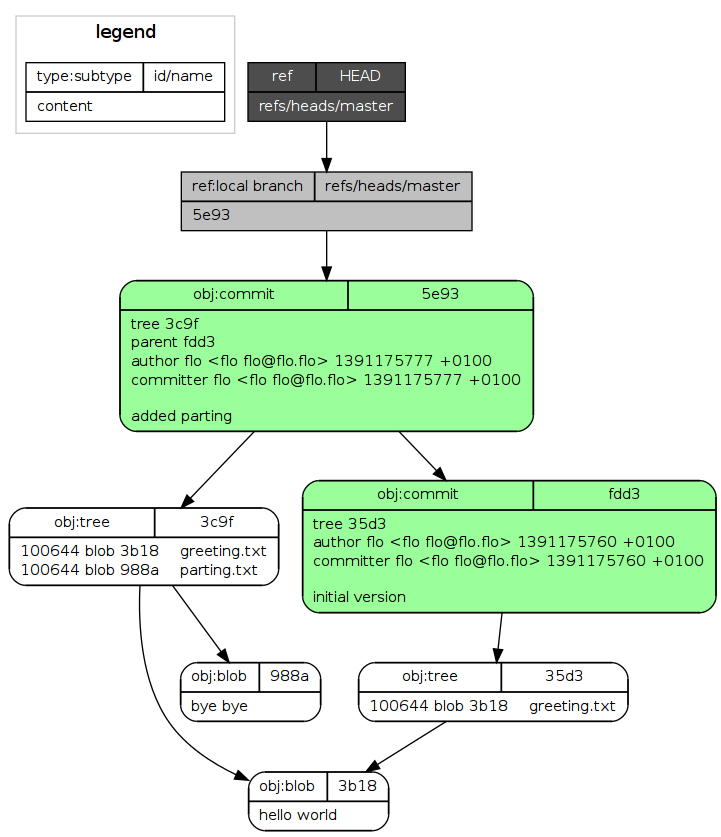
This document only teaches Git’s concepts. It does not present concrete examples how to work with Git. Read tutorials for that purpose.

## Overview

This chapter plunges you right into the guts of Git. Knowing the following, you can quickly deduce most of your questions arising in daily life using Git. The following is simple for an engineer. Investing the short time to understand these fundamental concepts pays a hundred fold.

*the data structures are really really really simple -- Linus Torvalds (Git’s author)*

Let’s start by giving a mental image on the text that follows. The following picture displays the content of a Git repository after 1) Created a file named greeting.txt with content “hello world” 2) Commited to Git, i.e. ordered Git to store a snapshot of all files and subdirectories. Currently this is a single directory, the project’s root directory, containing that single file, and the file itself. As commit message I chose “initial version” 2) Created a new file “parting.txt” with content “bye bye” 3) Commited to Git with commit message “added parting”:



A Git repository is essentially a database storing a set of Git objects. An object has an an id, an immutable content, and and implicit type (nowhere stored directly, but deducable). The id is a hash over the object’s content. The id universally uniquely identifies the object, in any repository in the universe. An object’s content can refer to zero or more other objects via their SHA1, resulting in a directed graph of objects. Possible object types are commit, tree, blob and tag. They are explained soon.

In addition to objects there are references. A reference has a type, a name, and content. The fully qualified name is hierarchically structured like the path in a file system, for example “refs/heads/master”. Typically you will see the abbreviated name, “master” in the case of the former example. The fully qualified name uniquely identifies the reference within a single repository only. I.e. references are a repository-local concept. The content of a reference is either the name of another reference or the SHA1 of a Git object. Possible types are branches and lightweight tags, branches being subdivided into local branches and remote tracking branches. They are explained soon.

The overall result is a DAG of references and Git objects, the references being the ‘entry points’ to the DAG of Git objects. From the following part explaining the object types more detailed can be deduced that the graph has no cycles. Git objects are only useful if they are reachable, i.e. when they are directly or indirectly referenced by a reference. Unreachable objects will eventually be deleted by Git’s garbage collector.

That’s it! You understand now Git at its fundamental core!

In practice you also need to know about the following Git object types, branches and tags. You’re not expected to understand all of this after reading it for the first time, but you should understand it eventually.

* Commit object: Represents a version of a file tree. I.e. the central part of a versioning tool – you want to store/freeze the current state of your files. The commit object’s content contains information about the committer, the commit message, the SHA1 of the tree object being the root of the project file tree, and a list of SHA1s to the parent commits of this commit. Typically a commit has one parent commit; a merge commit has typically two parent commits.
* Blob object (binary large object): Represents a file. The blob object’s content is the file content. Is always a leaf in the DAG.
* Tree: Represents a directory. The tree object’s content is a list of entries. Each entry contains the name of a file or subdirectory and the SHA1 of the blob or tree object representing it.
* Tag: Describes a Git object, typically a commit, with prose text. The tag object’s content is the SHA1 of the object being described, information about the author, and the message / description as text. Tags are typically used for releases, e.g. “R1.2.1”.

Branches and lightweight tags are references, reference being the more general concept. Branches are subdivided into local branches and remote tracking branches. These subtypes of references only differ in the way they are intended to be used. Using low level Git commands and not caring about conventions, you can do everything with every reference type.

* Lightweight tag: To eternally reference the same object, typically a commit or a tag object. Typically used to tag a commit as a release version, e.g. “R1.2.1”.
* Local branch: To add new commits locally.
* Remote tracking branch: A copy of a local branch of a remote repository.

That’s it really! You can now deduce and answer most questions people have which only read oversimplifying Git documentation.

Also medium-advanced Git concepts like the index (aka staging area), stashes and reflog are implemented directly using those concepts. Using the tool git-draw you can play with tiny Git repositories, displaying you the current content of the repository as a graph, inclusive the index and reflogs (which includes stashes). More information about git-draw is available on the Dragon wiki [here](http://swat/dragonwiki/index.php5/Git)

# Miscellaneous Basic Concepts

## Git tracks content

Each commit (other SCMs name that revision or version) stores the state of the whole file tree of the project. That includes naturally each file's content, but also directories’ content (i.e. filenames and directory tree structure). This is similar (but not identical) to what subversion does. In contrast, VSS, for each file, stores different versions. To 'freeze' a specific version of the whole file tree, labels have to be used in VSS.

Git only stores content, i.e. snapshots of the project's file tree. Git doesn't store changes. In particular, it doesn't store file renames / moves. Git's database internally might still try to safe disk space by storing only the delta between two files. But that's only about saving space and it's a purely internal thing deep down; it can do that for logically completely unrelated files, when they happen to be very similar.

For example see Figure 1. The project starts with the first commit a. The root directory's content is the file main.cpp. In the next commit b, the project root directory's content are the (sub-) directories source and include, which in turn contain the files main.cpp and main.h respectively. The file main.cpp has another content. In the next commit c, all directory’s content remained the same, only the file main.cpp has another content.

a

b

main.cpp

source/

include/

main.cpp

main.h

int main() {return 42;}

int main() {return 42;}

int answer() {return 42;}

int answer();

b

source/

include/

main.cpp

main.h

int main() {return answer();}

int answer() {return 42;}

int answer();

Commit, pointing to its content - the root dir of the project

Directory (aka tree), within its content – the list of files / subdirs

File (aka blob), within its content

Project’s file tree at commit a

Project’s file tree at commit b

Project’s file tree at commit c

Figure 1 Each commit (aka revision) stores the full project file tree at that revision.

The point to remember here is that each commit represents a certain state of the whole file tree.

Per project there is only a single .git directory at the root of the project, which contains all the information Git needs, most of all the whole history of the project.

When you paid attention in school you remarked that the drawing in Figure 1 is actually a DAG. More on that topic in in chapter …

## The working tree

The working tree (aka working directory) are the files and directories of your project you actually work with; the files which you edit. This is the same concept as in other SCMs such as SVN or VSS. The content of the working tree is based upon the HEAD commit (page 10). That normally is the current branch (page 14), see Figure 1.

Figure 2 Working tree is based upon the HEAD commit, here the current branch featureX.

The working tree is clean, if it corresponds to the commit / revision referenced by HEAD. Consequently a working tree is dirty if it contains modifications relative to the HEAD commit / revision.

## Nearly every operation is local and fast

Being a distributed system, no repository is special, every repository is as good as any other repository. Every repository has all the information there is at all, in particular the whole history of the project. Thus almost all operations are local (checkout, commit, branch, merge, view the whole history ...). Push and pull are one of the very few operations which need to talk to another repository, i.e. which potentially use the network.

An harddisk is fast compared to a network which connects your computer to a repository on a remote computer. Partly for that reason, all local operations are almost instantaneous.

# Miscellaneous small items

The intention of this chapter is to introduce and explain a few rather technical items. They are referenced by later chapters. The level of detail is too high for a first time reader; as a first time reader, you might just want to glimpse over this chapter.

## SHA1

A SHA1 is a 40 digit hex (160 bit) hash aka checksum. All git objects (see chapter Git objects / object database (page 8)) are identified with a SHA1, the commit object being the most notable.

A SHA1 id globally uniquely identifies a commit, or Git object in general. Globally means in any repository in the universe. If two commits in any repository anywhere have the same SHA1, that means they really are exactly the same thing. They represent exactly the same file tree and exactly the same history.

Within a given repository, usually not the whole SHA1 is required to uniquely identify something, but the first 6 digits or so are enough. Thus when working with Git, the full 40 digit id is seldom needed. Examples in books often use only short hex strings, say 4 to 6 digits long. An example of a full SHA1 is

24b9da6552252987aa493b52f8696cd6d3b00373

## References

Since branches are references, understanding references is a preliminary to understand branches. A reference is a pointer to a Git object (see chapter Git objects / object database (page 8)), typically a commit. In C++, the value of a pointer is the address of the objected pointed to. In Git the value of a reference is the SHA1 of the Git object it references.

A reference also can point to another reference, in which case it’s called a symbolic reference. The value of a symbolic reference is the name of the referenced reference. A notable example of a symbolic reference is HEAD, see chapter The current branch (page 14).

Figure 3 Reference and symbolic reference

## Git objects / object database

To be written

Git objects are: commit, tree (aka directory), blob (aka file) and tag.

# Commits

Each commit (other SCMs name that revision or version) stores the state of the whole file tree of the project. That includes naturally each file's content, but also directories’ content (i.e. filenames and directory tree structure). This is similar (but not identical) to what subversion does. In contrast, VSS, for each file, stores different versions. To 'freeze' a specific version of the whole file tree, labels have to be used in VSS.

## Commit id is a sha1

Git uses a SHA1 (see chapter SHA1 (page 7)) as id for a commit. In contrast, SVN or VSS use an incrementing integer as id for a given commit aka revision. In practice the SHA1 is less cumbersome as it first looks like. First, in daily work, commits are identified using branches, see chapter Branches (page 13). Second, as the chapter SHA1 explains, usually the first few digits, 6 or so, are good enough.

Why not use incrementing integers: because its distributed

how sha1s are built, i.e. they are a checksum over everything reachable from that object.

## Commit history is a DAG

Many Git documentations regard this as a quite advanced topic. I however think it is essential to understand well enough what a branch is, in order to easily work with branches.

A direct acyclic graph (DAG) is a graph where the edges have a direction and which contains no directed cycles. When you commit a new commit b on top of commit a, then b has a pointer to a. Commit a is the parent of b. Commit b is the child of a. Any commit reachable (by following the ‘parent pointers’) from a given commit c is an ancestor of c. Any commit which can reach a given commit d is a descendant of d. See Figure 4.

Figure 4 Building blocks of the commit DAG

A commit has zero or more children, and one or more parents. Each commit stores the pointers to its parents. It does not store pointers to its child; that would not be feasible, since children are added in the future. If a commit has more than one child, that means multiple lines of development branch away from this commit. If it has more than one parent, it is a merge commit, i.e. it is built by merging multiple lines of development together. To be precise, a commit can also have zero parents, in which case it is an initial commit of the repository. This results in an arbitrary complex DAG, see Figure 5.

Figure 5 A commit DAG can be arbitrary complex – well, according to the rules of a DAG

The bottom line to remember is: When two or more commits 'branch' away from a given commit that is just the nature of the commit DAG. A Git branch is something else, see chapter What is a branch? (page 14).

## Commit (action)

To be written

## Commit (revision)

To be written

## Checkout, switch

To be written

## Commit properties

To be written

## The HEAD commit

S is the current commit. SVN on the other hand means the latest & greatest revision in the (central) repo s

Figure 6 HEAD is a reference to a branch or commit, finaly defining the HEAD commit

# Index aka staging area

The index (aka **staging area** or **cache**) is what your next commit will look like. The purpose is to have a tool which lets you define / craft exactly how your next commit will look like. To add (aka stage) means adding (‘copying’) a file from the working tree to the index. To commit means turning the index into a new commit. There is also a way to edit the index’s content, e.g. only ‘copy’ parts of the modifications within one file.

Figure 7 Index overview

When you modify files and want to commit these modifications, you have to add them to the index first with the Git add command. For beginners that sounds cumbersome. However, first, the commit command has an option (-a) which automatically adds all modified files to the index before committing. Thus beginners still have what they want. Second, you will learn to love the index since it is a great tool to craft the next commit. One can modify files, add some (not necessarily all) of the changed files to the index, modify some more files, again add only some of the changes to the index etc. It’s even possible to add only parts of the changes within one file, or even entirely edit the content of a file in the index.

For example in Figure 8, in the ‘working’ row, the modifications to the files, and adding modifications to the index, where done in an arbitrary order. Note that in the 3rd file, only parts of the modification where added to the index using the ‘edit index’ feature. Once committing, the figure shows that the new commit is identical to the index. Note that not all modifications of the working tree where committed (only those in the index); naturally you still have them and can commit them later in a further commit.

The index is also used by Git during merges, in particular for merge conflicts. See chapter Merge conflicts (page 17).

Further reading: chapter [the index](http://www.kernel.org/pub/software/scm/git/docs/user-manual.html" \l "the-index) (Git User's Manual), [index](http://www.kernel.org/pub/software/scm/git/docs/gitglossary.html" \l "def_index) in (gitglossary(7) Manual Page), [git-add](http://www.kernel.org/pub/software/scm/git/docs/git-add.html) in (git(1) Manual Page), [git-commit](http://www.kernel.org/pub/software/scm/git/docs/git-commit.html) in (git(1) Manual Page), chapter [2.2 Git Basics - Recording Changes to the Repository](http://git-scm.com/book/en/Git-Basics-Recording-Changes-to-the-Repository) in (Chacon)

Figure 8 Index detailed

# Tags

A tag is a reference (aka pointer, see chapter References (page 8)) to a Git object, typically a commit. Tags are mostly used to mark important commits in the history, e.g. most commonly releases, e.g. v2.1.0. Thus they can be used to give a commit which is otherwise identified by a SHA1 a human friendly name. There are two flavors of tags: lightweight tags and annotated tags.

A lightweight tag is really only a reference. It is thus very much like a branch, only that it is intended to statically reference the same commit (or any Git object) eternally. Contrast this with a branch, which is also a reference, but which changes the commit it refers to. E.g. when committing to the current branch, the current branch refers to the new commit after the operation.

An annotated tag has in addition a tag object. A tag object has further information such as a message, author and date. The message is used to describe the referenced commit, e.g. “V2.1.0: Added features are … Added bug fixes are …”. So in the case of an annotated tag, the tag reference refers to the tag object, which in turn refers to the actual commit object (or any object).

Further reading: command [tag](https://www.kernel.org/pub/software/scm/git/docs/git-tag.html) in (git(1) Manual Page), chapter [Git Basics - Tagging](http://git-scm.com/book/en/Git-Basics-Tagging) in (Chacon)

# Branches

## What is a branch?

The term branch is often used for two similar things.

The term **branch** is often used as synonym for branch head. The term branch head (or just head) means a reference (aka pointer, see chapter References (page 8)) to the most recent commit on a branch, which is called tip of branch.

When we need to be precise, the term branch means a line of development. The tip commit and its ancestors commits are said to be ‘on’ (or part of) that branch.

Figure 9 The term ‘branch’ has multiple interpretations

For almost all Git operations it is best to see a branch as a reference to a commit. That’s also how it is technically internally implemented. That makes branches extremely lightweight. In fact, a branch is just a file (explore myproject/.git/refs/heads/...) with the same name as the branch, which’s content is the 40 byte SHA1 of the referenced commit.

Creating a branch is just creating a reference to a commit – the one you want to branch away from. Committing to a branch automatically changes the pointer so the branch points to the new commit.

Figure 10 Creating a branch called featureX based on commit a just creates a pointer pointing to commit a. Committing to the branch featureX lets it point to the new commit.

## The master branch

A Git repository by default starts with a branch called master. There is nothing special at all about that branch. It can be deleted, as any other branch, if you like. However most projects use it for the main line of development.

## The current branch

A Git repository (almost) always has one current branch. In many GUI it is displayed highlighted in contrast to the other branches, often by using a bold font,. Many Git operations operate on it. More precisely, most of these operations operate on the HEAD commit. If a repository is in the so-called detached head state there is no current branch. See also chapter The HEAD commit (page 10).

Figure 11 The current branch, here featureX, is defined by the HEAD reference.

## Deleting branches or commits

Deleting a branch just means deleting a pointer, since a branch is just a pointer. From a first point of view, no commits are deleted. Only if deleting the pointer results in unreferenced commits, the unreferenced commits will be deleted by the garbage collector sooner or later.

Figure 12 shows an example in which no unreferenced commits result from deleting a pointer. Thus nothing really happens apart from deleting the pointer. The shown starting situation is typical for the situation after a merge, see chapter Merging (page 16).

Figure 12 Deleting a branch

Figure 13 shows an example in which unreferenced commits result. The shown starting often results when one worked on a branch, did not yet merge it, and wants to forget about the work done on that branch.

In this example that results in commit b becoming unreferenced, i.e. nothing is directly or indirectly referencing it. It is thus 'deleted'. To be precise, for the time being it's just 'invisible'. It is only really deleted by the garbage collector at some time in the future.

Figure 13 Deleting a branch resulting in unreferenced commits

To undo the deleting of such a branch is a bit cumbersome, since the commit it referenced has become unreferenced and is thus no longer visible. In such a case the reflog is your friend, see

(git(1) Manual Page)

## 

(git(1) Manual Page)(Git User's Manual)(Chacon)

To be written (page 29).

# Merging

Merging means creating a new commit, called a merge commit, which has multiple parent commits. Typically it’s two parents. If it has more than two, it is called an octopus merge. An alternative command to merge with similar result is rebase, see chapter Rebase (page 19).

## Automatic merge with no merge conflicts

The following assumes that the merge can automatically be done, i.e. that there are no merge conflicts. Figure 14 shows merging branch featureX into branch featureY. Typically that is done by making featureY the current branch and then execute the merge command with featureX as argument. A new commit is created which has two parents – the commits the branches pointed to right before the merge. The branch featureY points to the new commit while the branch featureX remains unchanged.

Figure 14 Merging branch featureX into the current branch, here featureY, creates the merge commit ace5.

The commit featureY pointed to right before the merge is called the first parent. It is not special to Git; it’s only that some GUIS showing the history have the sometimes convenient feature of showing a commit history only following the first parent pointers.

A merge – that is a concept which has nothing directly to do with Git - always needs 3 files / version: 1) the base revision 2) the revision containing all changes made by person 1 3) the revision containing all changes made by person 2. Git automatically finds the commit which serves as a merge base, i.e. the common ancestor of the two commits to be merged together. Git does that by analyzing the commit DAG. For simple graphs such as the one in Figure 14, this is trivial. However for arbitrary complex DAG, e.g. see Figure 5 (page 6), that can be a non-trivial task. It’s the strength of Git that it nevertheless can do it.

## Merge conflicts

Git is very good at doing the merge automatically and fast. That automatism is only possible if the two lines of development did only independent changes which do not conflict. However, if they changed for example the same line in the same file, there is no way Git could automatically know which of the two versions of that line it should take.

If Git detects a merge conflict, it cannot complete the automatic merge, i.e. it does not create a merge commit. It does not change the current branch. It creates the reference MERGE\_HEAD pointing to the other branch. To be continued …

Figure 15 Situation after a failed automatic merge (see Figure 14 for the starting state) due to merge conflicts

Abort a merge … to be written

## Fast forward merge

Say featureY is the current branch, and you want to merge branch featureX. If the current branch is an ancestor of featureX that is called a fast forward merge. The featureY can just forward its pointer to the same commit as featureX. This is because there is no divergent work, featureY being an ancestor of featureX. Only featureX did changes, featureY did no changes at all. Merging ‘no changes’ with ‘any changes’ means simply taking ‘any changes’.

In a true merge, a merge commit is always created.

Figure 16 Merging branch featureX into the current branc, featureY.Feature Shows the difference between a fast forward merge and a true merge.

## Squash merge

To be written

Figure 17 Squash merge

## Merge concepts

Many people seem to think that in a merge process, dates are somehow involved. That is not the case. A merge – a concept independent of Git - always has three operands: the common base version B, version 1, version 2. A merge is the task of creating a new version M, where the difference B to M is the union of the difference B-1 and B-2. I.e. version M contains the work / changes of both sides.

Figure 18 Merge

So when Git is asked to merge commit 1 and commit 2, it first has the find a commit which can serve as merge base. It does that by looking at the commit DAG. See also the manual page of [git-merge-base](http://www.kernel.org/pub/software/scm/git/docs/git-merge-base.html). To emphasize again – that is purely a graph theory task, dates are not involved in any way. Now Git has everything it needs to know to merge: a common base, version 1 and version 2.

# Rebase

To be written

# Changing History / Fixing Mistakes

A commit is identified by a SHA1. The sha1 is a checksum of everything that makes up that commit: all its parents and their parents (i.e. the whole history reachable from that commit). The content of the whole file tree for each commit. The commit message, author, date … . Thus if anything of that would change, that would also change the SHA1, i.e. the id, i.e. you really would have another commit object. Thus nothing of a given commit can be changed. However, you can 'change' it by replacing it with a new commit which contains the desired changes.

To be written

Rebasing is also changing history, it creates new commits, it doesn’t just move pointers of the commit DAG. You’ll notice new sha1 ids for the new commits.

## Amend a commit

To be written

# Distributed Git

Git is a distributed system. Each repository can talk to multiple other repositories. Technically, each repository is absolutely equal. In particular, technically, there is no central repository.

Everybody has it’s own local repository. The communication to one’s one repository is basically the same as it is in a central/traditional version control system, see Figure 19. It’s still commit (aka checkin) and checkout. Before the repository was typically on a remote server, now it’s typically on the local computer. What is new are commands to exchange data between repositories.

Figure 19 basic centralized VCS commands vs. basic distributed VCS commands

The basic operations to exchange commits between repositories are push and fetch, see Figure 20. When A pushes to B, that means commits are transferred from A to B. When B fetches, commits are transferred from B to A. Another common operation is pull, which is a fetch followed by a merge, where the remote tracking branch of the current branch is merged into the current branch. That will become clear later. If one starts contributing to an existing project, she typically wants to have her own repository to work with, so she clones it from an existing repository, to get her own local clone aka copy.

Figure 20 Push, fetch, pull

## Distributed workflows

As said, technically each repository is absolutely equal, no one is special. However the users can think of certain repositories as special and develop a workflow accordingly. For example a possible setup, commonly known as ‘centralized workflow’ is one which closely resembles a traditional version control system: There is one central (aka blessed) repository. Each developer has its own repository. Each developer fetches-to / pushes-from the central repo. More details, and more common workflows can be found [here](http://git-scm.com/book/en/Distributed-Git-Distributed-Workflows).

## Clone

Say Sally makes a clone from a repository. In the default configuration, that gives her all the commits that are in the remote repo, all the tags that are in the remote repo and a remote tracking branch for each branch in the remote repo. See Figure 21 for an example. The yellow remote tracking branch origin/working is explained in the next chapter.

Figure 21 Clone

## Remote tracking branch

A remote tracking branch is a regular branch used to follow changes from another repository. In the example below, that is ‘origin/working’. The first part, here ‘origin’, is the name of the remote repo from Sally’s repo point of view. ‘origin’ is the default name used for the remote repo when you clone. If Sally would fetch now from an even further repo which she calls ‘bob’, then she would have new remote tracking branches ‘bob/working’. Maybe also a ‘bob/foo’ if bob has a branch foo.

The remote tracking branch does not continuously automatically follow what happens on the remote repo. As long as Sally does not fetch-from or pushes-to it, Sally’s origin/working will stay where it is, independent of what happens on the remote repo. After all, in a distributed system we want to talk to the other repo, i.e. use the slow network which might not always be available, as seldom as possible.

Say Sally now starts working and makes a new commit (ID c) on her working branch. In the meantime, somebody else also makes a new commit (ID 1) on the working branch of the remote repo. Figure 22 shows the new situation. The branch featureX is used in the next chapter.

Figure 22 Remote tracking branch & upstream branch

## **Upstream (aka tracking) branch**

When a branch X has an upstream branch (aka tracking branch), that makes using pull and push more convenient. Pull and Push will be explained in the next chapters, so you may want to return here later when you do not yet understand push and pull. Typically the upstream branch is a remote tracking branch.

Say Sally creates a local branch featureX, intending to work for the working branch in the remote repo, see Figure 22. She is not using her local working branch because she possibly has many more local feature branches. Working on featureX, she eventually has to push or pull. If she would not specify origin/working as the upstream branch of featureX, she always had to specify the operands featureX and origin/working to the push and pull command respectively. But when she defines origin/working to be the upstream branch of featureX, the push and pull commands are happy with only the operand featureX – they now can deduce themselves that the 2nd operand is origin/working.

Using the SmartGit GUI, defining an upstream branch is even required to be able to push or pull. There is no other way of defining the 2nd operand for push/pull.

## Fetch

A fetch, similar to a clone, gives her all the commits of the remote repo, all the tags of the remote repo, and a remote tracking branch for all the branches in the remote repo. To be precise, ‘all’ is only by default, if you want you can specify what you want to get and under which reference names you want to store it locally. Naturally Git is smart enough to only transfer the data Sally doesn’t have already. Since typically new branches are seldom added to the remote repo, that typically means Sally’s remote branches are updated and point now to new commits.

Say Sally now starts working and makes a new commit (ID c) on her working branch. In the meantime, somebody else also makes a new commit (ID 1) on the working branch of the remote repo. Also a new branch R8.0 with a new commit (ID 11) was created. Sally doesn’t know that yet, but to find out what the current state of the remote repo is, she makes a fetch.

Figure 23 Fetch

## Pull

Pull is a fetch followed by a merge. With the default configuration, the upstream branch of the current branch is merged into the current branch.

Figure 24 Pull

## Push

A push is somewhat the inverse of a fetch, just in the opposite direction: it also transfers commits, it also updates branches (i.e. let references point to new commits) in the destination repo and it also does not merge anything (i.e. it does only fast-forward merges). However it only pushes one branch which has to be specified and which has to have a remote tracking branch - at least this is so in the default configuration. So if Sally pushes her working branch, Git knows origin/working is the remote tracking branch of it, so in the remote repository called ‘origin’ the branch ‘working’ will point to the received commits.

So Sally merges origin/working into her working, creating a merge commit, in the example with ID d, see Figure 25. (Note that in the Dragon Git workflow we do *not* work that way – we allways work on feature branches and merge the result into working, which we then push!). To show another property of push which will be later explained, say Sally also created a new branch foo and commit a new commit to it.

She can now push, which by default pushes the current branch, in this example working. That transfers the commits c and d to the remote repo. In the remote repo, the branch working is fast-forward merged from commit 1 to commit d. In Sally’s repo, the remote tracking branch origin/working also reflects the state of working in the remote repo. Note that in this example Sally only pushed using the defaults, i.e. she pushed the current branch, i.e. working, i.e. she did not push others of her branches such as her new foo branch.

Figure 25 Push

Further info for the interested reader: Why push must be a fast forward? Say Sally is now finished with her work and wants that the remote repo also contains her work, so she would like to push her working branch. Before she can do that, she must merge her working and the origin/working. If she would not do that and just push working as it is in Figure 23 (before or after the fetch does not matter), the push would fail. It needs to fail, because in the remote repo, the working branch can only either point to Sally’s new commit (ID c), or the new commit from the other person (ID 1). If the push would have succeeded, that meant that in the remote repo working would point to Sally’s commit (ID c), and the commit (ID 1) would become unreferenced and thus lost. Automatically merging Sally’s and the other person’s commit within the remote repo is not possible. Probably because for merging, or at least for resolving a merge conflict, a working tree is needed. There are so called bare repositories which don’t have a working tree. Also in case of merge conflict, Sally would have to work with the working tree of the remote repo – that’s apparently a thing which Git’s designer didn’t want. Everybody should work only with it’s own working tree, never with the working tree of another repo.

# Miscellaneous Advanced Concepts

## The stash

Use git stash when you want to record the current state of the working directory, the index and optionally untracked files, but want to go back to a clean working directory. The command saves your local modifications away and reverts the working directory to match the HEAD commit. You later can apply the stash again to get back what you stored. You can apply any stash entry to any commit.

The stash is useful for example if you’re in the middle of a task, and now quickly need to do something else, e.g. a bugfix.

It’s also useful when you made too many modifications to your working tree, and want to make multiple commits out of it, and also want to run the test suite on each ‘commit’ before actually committing. You then start by adding the first set of modifications into the index, run stash with the option ‘keep index’, then run the test suite, and if green, make the commit. Then you apply the stash, and continue the process.

Technically the stash is just the reflog of the stash reference. Each entry in the reflog is a stash entry. It refers to a commit holding the stored working tree. That commit has as parents the HEAD and a 2nd commit which holds the stored index.

Further reading: chapter [Temporarily setting aside work in progress](http://www.kernel.org/pub/software/scm/git/docs/user-manual.html" \l "interrupted-work) in (Git User's Manual), [git-stash](http://www.kernel.org/pub/software/scm/git/docs/git-stash.html) in (git(1) Manual Page), chapter [6.3 Git Tools - Stashing](http://git-scm.com/book/en/Git-Tools-Stashing) in (Chacon)

## Git has integrity / trust

Is actually not an ‘advanced’ concept but a very fundamental property of Git.

…. That enables Git to ensure the integrity of the database. It's impossible that corruption of the data (due to memory, hard disk, transfer over the net or even accidentally malicious …) goes undetected. Integrity is implemented at Git's core and is a very fundamental part of its philosophy.

Further reading: [Trust](http://www.kernel.org/pub/software/scm/git/docs/user-manual.html" \l "trust) in (Git User's Manual), [Git Basics - Git Has Integrity](http://git-scm.com/book/en/Getting-Started-Git-Basics" \l "Git-Has-Integrity) in (Chacon)

## Blame (aka annotate or praise)

Each line in a file is annotated with the information, which commit (e.g. who and when) changed that line. To be written

Further reading: [git-blame](http://www.kernel.org/pub/software/scm/git/docs/git-blame.html) in (git(1) Manual Page), chapter [File Annotation](http://git-scm.com/book/en/Git-Tools-Debugging-with-Git" \l "File-Annotation) in (Chacon)

## Bisect

Find by binary search the commit that introduced a bug. To be written

Further reading: command [git-bisect](http://www.kernel.org/pub/software/scm/git/docs/git-bisect.html) in (git(1) Manual Page), chapter [How to use bisect to find a regression](http://www.kernel.org/pub/software/scm/git/docs/user-manual.html" \l "using-bisect) in (Git User's Manual), chapter [Binary Search](http://git-scm.com/book/en/Git-Tools-Debugging-with-Git" \l "Binary-Search) in (Chacon)

## Pickaxe

To find a commit changing/adding/removing a line containing a given string/regex. I.e. with that you can follow a method moving within a file or even across files. To be written

Further reading: options -S and -G in command [git-diff](http://www.kernel.org/pub/software/scm/git/docs/git-diff.html) in (git(1) Manual Page), [pickaxe](http://www.kernel.org/pub/software/scm/git/docs/gitglossary.html" \l "def_pickaxe) in (gitglossary(7) Manual Page)

## The full DAG

Actually not only the commit history is a DAG. Git’s DAG is actually bigger. contains a few concepts, such as tags or branches or references, which are not introduced yet. Don’t worry, just remember that actually everything is a DAG made out of Git objects.

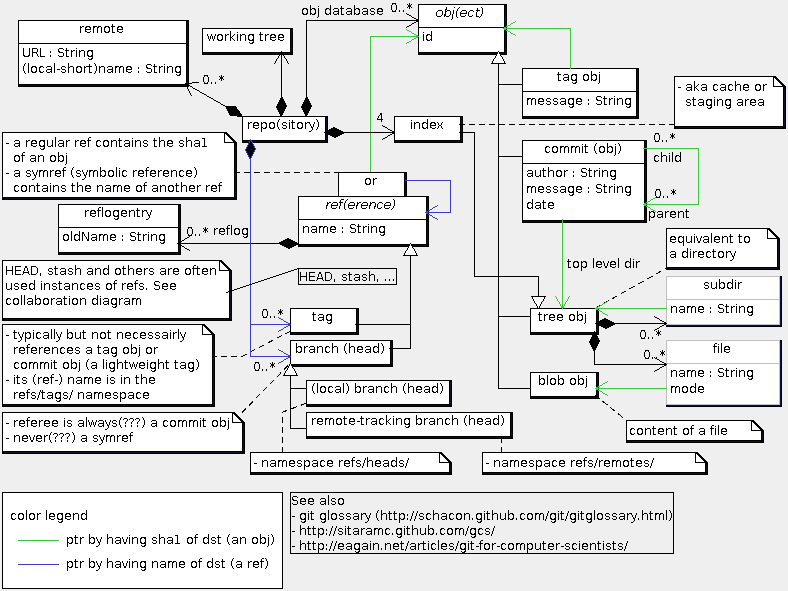
Figure 26 Example of Git’s DA

The following is just a quick excursion. More details can be found here: <http://eagain.net/articles/git-for-computer-scientists/>. A commit also stores a pointer to the file tree which represents the file tree of your project at that version. A tree (aka directory) stores pointers to its sub trees and to blobs (aka files) it contains. So you end up with a DAG consisting of commits, trees, blobs, branches and more.

Further reading: [DAG](http://www.kernel.org/pub/software/scm/git/docs/gitglossary.html" \l "def_DAG) in (gitglossary(7) Manual Page)

## Git as UML class diagram

This UML class diagram shall serve as an overview of how Git concepts fit together. The source can be found at (Kaufmann)



## Garbage collector

To be written

Further reading: commands [git-gc](http://www.kernel.org/pub/software/scm/git/docs/git-gc.html), [git-prune](http://www.kernel.org/pub/software/scm/git/docs/git-prune.html) and [git-fsck](http://www.kernel.org/pub/software/scm/git/docs/git-fsck.html) in (git(1) Manual Page), chapter [9.7 Git Internals - Maintenance and Data Recovery](http://git-scm.com/book/en/Git-Internals-Maintenance-and-Data-Recovery)

## Pack files

To be written

Further reading: command [git-repack](http://www.kernel.org/pub/software/scm/git/docs/git-repack.html) in (git(1) Manual Page), chapter [How git stores objects efficiently: pack files](http://www.kernel.org/pub/software/scm/git/docs/user-manual.html" \l "pack-files) in (Git User's Manual),chapter [9.4 Git Internals - Packfiles](http://git-scm.com/book/en/Git-Internals-Packfiles) in (Chacon)

## Reflog

Each reference (followinly each branch) has a log where it pointed to. To be written

Further reading: command [git-reflog](http://www.kernel.org/pub/software/scm/git/docs/git-reflog.html) in (git(1) Manual Page), chapter [Reflogs](http://www.kernel.org/pub/software/scm/git/docs/user-manual.html" \l "reflogs) in (Git User's Manual), chapter [RefLog Shortnames](http://git-scm.com/book/en/Git-Tools-Revision-Selection" \l "RefLog-Shortnames) in (Chacon)

## No locking of files

Git has no concept of ‘locking’ a file to prevent others from changing it. To be written

## No explicit file moving / renaming

To be written

## Attributes

To be written

Further reading: chapter [7.2 Customizing Git - Git Attributes](http://git-scm.com/book/en/Customizing-Git-Git-Attributes) in (Chacon)

## Grafts

To be written

Further reading: item grafts in [gitrepository-layout(5)](http://www.kernel.org/pub/software/scm/git/docs/gitrepository-layout.html) in (git(1) Manual Page), [grafts](http://www.kernel.org/pub/software/scm/git/docs/gitglossary.html" \l "def_grafts) in (gitglossary(7) Manual Page)

## Replacing Objects

To be written

Further reading: command [git-replace](http://www.kernel.org/pub/software/scm/git/docs/git-replace.html) in (git(1) Manual Page)

## Patch-Id

To be written

## Hooks

To be written

# Appendix

## Glossary

See (gitglossary(7) Manual Page)

## Index

add 8

ancestor 7

branch 11

branch head 11

delete 13

master branch 12

**cache** 8

clone 18, 19

commit

child 7

merge commit 14

parent 7

current branch 12

DAG *See* direct acyclic graph

descendant 7

direct acyclic graph 7, 25

fetch 18, 20

first parent 14

head *See* branch head

index 8

merge

Fast forward merge 15

merge base 15

octopus merge 14

true merge 15

MERGE\_HEAD 15

Merging 14

pull 18

Pull 21

push 18, 22

remote tracking branch 19

SHA1 5, 6

stage 8

**staging area** *See* index

stash 24

tag 11

annotated 11

lightweight 11

object 11

tip of branch 11

tracking branch 20

upstream branch 20

working directory *See* working tree

working tree 4

clean 5

dirty 5

## Bibliography

Chacon, S. (n.d.). *Pro Git.* http://git-scm.com/book/en.

*Git User's Manual.* (n.d.). Retrieved from http://www.kernel.org/pub/software/scm/git/docs/user-manual.html

*git(1) Manual Page.* (n.d.). Retrieved from http://www.kernel.org/pub/software/scm/git/docs/

*gitglossary(7) Manual Page.* (n.d.). Retrieved from http://www.kernel.org/pub/software/scm/git/docs/gitglossary.html

*gittutorial(7) Manual Page.* (n.d.). Retrieved from http://www.kernel.org/pub/software/scm/git/docs/gittutorial.html

Kaufmann, F. (n.d.). *Git UML docu, created with ArgoUML.* Retrieved from \\swsources.esec.com\GIT\flka\gitconnector\git\_docu.uml