vogella series

vogella

Distributed Version Control with Git

Mastering the Git command line - Third Edition

Lars Vogel

Foreword by Alex Blewitt

Distributed Version Control with Git

Table of Contents

Distributed Version Control with Git

Mastering the Git command line

Lars Vogel

14.12.2014

Dedication

For my parents Otfried and Mira

Foreword

Version control systems have been used to store source code and documents for most of the last forty years. With the growth of open source projects, version control systems have become commonplace. From the evolution of SCCS to RCS and CVS, and complete rewrites like SVN, version control has evolved with each being an incremental improvement over the former.

Distributed version control systems (DVCS) are the next leapforward in that evolution, and are rapidly displacing the incumbents, with an estimated 50% of projects now using DVCS in 2014. With Git as the clear winner within the DVCS, Git has become the number one version control system.

Although there are many distributed version control systems (including commercial solutions such as BitKeeper), Git stands head and shoulders above them. Older solutions, such as Darcs and Arch are no longer significantly developed or used, and competitors such as Bzr are far behind in adoption. The only close competitor to Git is Mercurial (also known as Hg), but with Google Code now supporting both, the majority of new projects are choosing Git.

This book introduces the Git distributed version control system, and provides the basics for getting started, from initializing a repository to committing, branching, pushing and pulling. It covers concepts such as the differences between distributed and centralized version control systems and provides links to Git hosting providers. After reading this book you should be able to perform all of the basic steps necessary to create and host your own Git-based repositories.

Lars's skills in explaining and presenting new concepts have been honed with his vast set of tutorials at vogella.com, as well as his training courses and conference presentations. This book is highly recommended for those who are new to either Git or DVCS.

Dr Alex Blewitt, Founder of Bandlem.com

Preface

1. Welcome

The Git version control is currently the de facto standard for a distributed version control system. It is used by several popular open source projects like the Linux kernel, the Eclipse IDE or theAndroid mobile operating system as well as within large IT companies like SAP AG or Google.

This book explains the usage of the Git version control system via the command line.

Thanks for getting this book. Most of the content of the book is also available online, but I hope that having this material available offline will help you to work more efficiently.

I hope you enjoy your learning experience.

2. Prerequisites

This book assumes that you already have experience in using a command line shell. It does not require any previous knowledge in programming or working with other version control systems.

The examples in this book were developed and tested on Linux. They should work also on Mac OS X. Windows users have to modify the shell commands slightly, e.g., instead of "ls" or listing the directory they need to use the "dir" command.

In case you selected this book in order to learn about programming you should start with a different reference first. This book is only about the Git version control system.

3. Errata

Every book has errors/mistakes to a certain degree. You can find a list of the known bugs on the errata page of the vogella.com website. The URL to this page is http://www.vogella.com/book/git/errata.html

In case you find errors which have not yet been reported, please let me know under: erratabooks@vogella.com.

Errors/mistakes might be one of the following:

- Typographical errors
- Examples that do not work as described in the book
- Factual errors that cannot be open to interpretation

4. How this book is organized

This books starts with an introduction into distributed version control systems. It continues to describe the basic Git terminology and how you can configure your Git tools.

If you are new to Git you may want to read Part I, “Introduction to Git and distributed version control” first, then jump to Part III, “Installation and configuration” and run through the Part V, “Exercise: A simple local Git workflow” to learn how to use the Git tooling for a local workflow.

As the book advances you learn how to connect to remote repositories and how to use branches and tags. The book covers merging and rebasing changes and provides all the necessary tips and tricks to use Git.

It also covers the usage of the popular online Git hosting platforms GitHub or Bitbucket and describes typical Git workflows which are considered as good practice.

5. About the author - Lars Vogel

Lars Vogel is the founder and CEO of the *vogella GmbH* company. He is one of the core developers of the Eclipse platform project and loves to share his knowledge by writing online tutorials and books. He is also a regular speaker at international conferences.

For the customers of the *vogella GmbH* he delivers development, consulting, coaching and training in the areas of Eclipse, Android and Git. These customers include Fortune 100 corporations as well as individual developers.

Lars is a nominated *Java Champion* since 2012. In 2010 he received the *Eclipse Top Contributor Award* and in 2012 the *Eclipse Top Newcomer Evangelist Award*.

6. About the vogella company

The *vogella GmbH* company offers expert development, consulting, coaching and training support in the areas of Eclipse, Android and Git. See training offerings and implementation support for details.

With more than one million visitors per month vogella.com is one of the central sources for Eclipse, Java and Android programming information.

7. Acknowledgments

The creation of this book did not follow the typical book creation process. It started as (and still is) a tutorial on my webpage.

A big "thank you" goes to Robin Stocker and Alex Blewitt for reviewing the third edition. I thank Matthias Sohn, Dirk Fauth, Dariusz Luksza, Thanh Ha, Stefan Lay and Krzysztof Daniel for the extensive feedback on the second edition.

A huge thanks also to Dr Alex Blewitt for writing the foreword and providing feedback on the content.

I got many suggestions or corrections from my readers and other people involved with Git and I would like to express my deepest gratitude to their contribution.

Part I. Introduction to Git and distributed version control

Chapter 1. Distributed version control systems

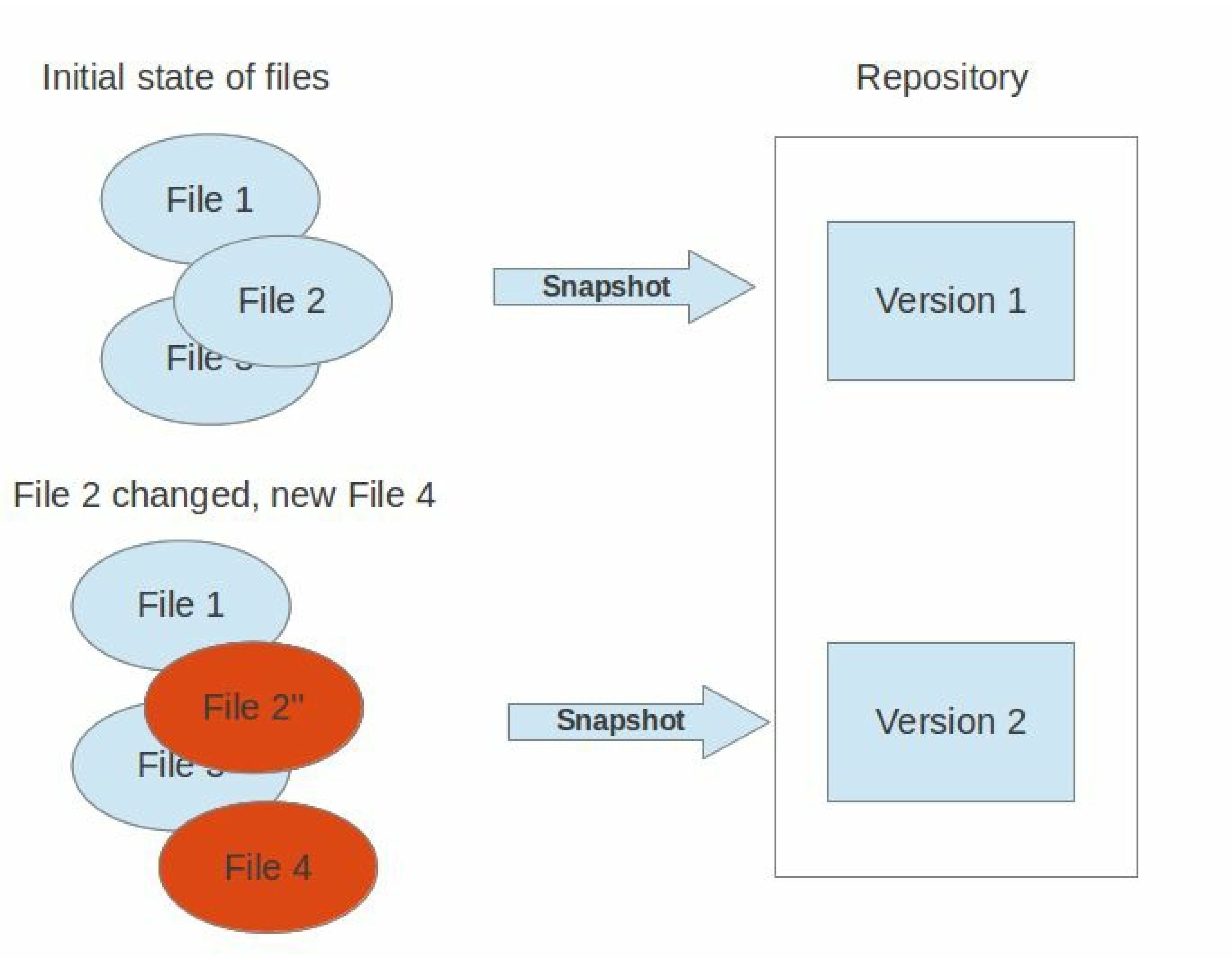
The following chapter gives an overview of what Git is about and how it works.

1.1. What is a version control system?

A version control system allows you to track the history of a collection of files and includes the functionality to revert the collection of files to another version. Each version captures a snapshot of the files at a certain point in time. The collection of files is usually *source code* for a programming language but a typical version control system can put any type of file under version control.

The collection of files and their complete history are stored in a *repository*.

The process of creating different versions (snapshots) in the repository is depicted in the following graphic. Please note that this picture fits primarily to Git. Other version control systems like *Concurrent Versions System* (CVS) don't create snapshots but store file deltas.



These snapshots can be used to change your collection of files. You may, for example, revert the collection of files to a state from 2 days ago. Or you may switch between versions for experimental features.

1.2. What is a distributed version control system?

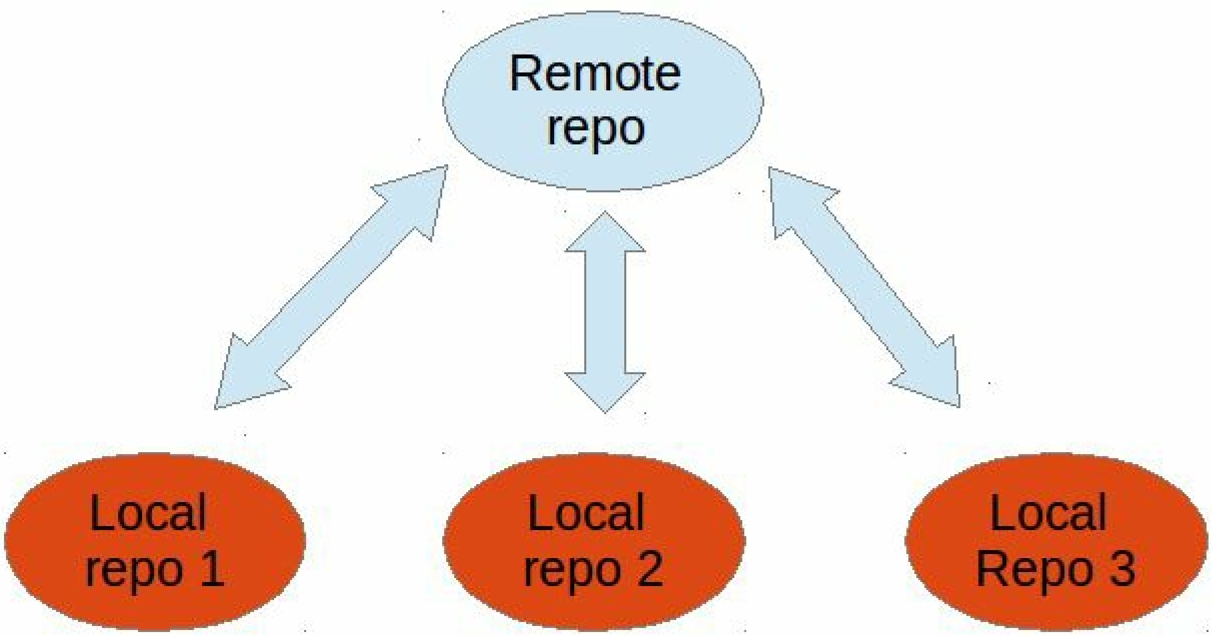
A distributed version control system does not necessarily have a central server which stores the data.

The user can copy an existing *repository*. This copying process is typically called *cloning* in a distributed version control system and the resulting repository can be referred to as a *clone*.

Typically there is a central server for keeping a repository but each cloned repository is a full copy of this repository. The decision which of the copies is considered to be the central server repository is pure convention and not tied to the capabilities of the distributed version control system itself.

Every *clone* contains the full history of the collection of files and a cloned repository has the same functionality as the original repository.

Every repository can exchange versions of the files with other repositories by transporting these changes. This is typically done via a repository running on a server which is, unlike the local machine of a developer, always online.



1.3. What is Git?

Git is a distributed version control system.

Git originates from the Linux kernel development and is used by many popular open source projects, e.g., the Android or the Eclipse developer teams, as well as many commercial organizations.

The core of Git was originally written in the programming language *C*, but Git has also been re-implemented in other languages, e.g., Java, Ruby and Python.

Chapter 2. Tools

2.1. The Git command line tools

The original tooling for Git is based on the command line, i.e. the Git development team provides only tooling for the command line.

This description is based on the Git command line tooling which offers all capabilities of Git.

2.2. Separating parameters and file arguments in Git commands

The double hyphens (--) in Git separates out any references or other options from a path (usually file names).

Using this allows you for example to distinguish between looking at a file called HEAD from a Gi commit reference called HEAD.

In case Git can determine the correct parameters and options automatically the double hyphens can be avoided.

```
# seeing the git log for the HEAD file
git log --HEAD

# seeing the git log for the HEAD reference
git log HEAD --

# if there is no HEAD file, the second command becomes
git log HEAD
```

2.3. Graphical tools for Git

You can also use graphical tools see GUI Clients at the official git website for an overview.

For example the Eclipse IDE provides excellent support for working with Git repositories.

To learn more about the Git integration into Eclipse see the Eclipse Git online tutorial or the Eclipse IDE book.

Chapter 3. Introduction to repositories and branches

3.1. Local repository and operations

After cloning or creating a repository the user has a complete copy of the repository. The user performs version control operations against this local repository, e.g., create new versions, revert changes, etc.

You can configure your repository to be a bare or a non-bare repository.

- bare repositories are used on servers to share changes coming from different developers
- non-bare repositories allow you to create new changes through modification of files and to create new versions in the repository

If you want to delete a Git repository, you can simply delete the folder which contains the repository.

3.2. Remote repositories

Git allows the user to synchronize the local repository with other (remote) repositories.

Users with sufficient authorization can *push* changes from their local repository to remote repositories. They can also *fetch* or *pull* changes from other repositories to their local Git repository.

3.3. What are branches in Git?

Git supports *branching* which means that you can work on different versions of your collection of files. A branch separates these different versions and allows the user to switch between these versions to work on them.

For example, if you want to develop a new feature, you can create a branch and make the changes in this branch without affecting the state of your files in another branch.

Branches in Git are local to the repository. A branch created in a local repository, which was cloned from another repository, does not need to have a counterpart in the remote repository. Local branches can be compared with other local branches and with *remote-tracking branches*. A remote-tracking branch proxies the state of a branch in another remote repository.

Git supports the combination of changes from different branches. This allows the developer, for example, to work independently on a branch called *production* for bugfixes and another branch called *feature_123* for implementing a new feature. The developer can use Git commands to combine the changes at a later point in time.

For example, the Linux kernel community used to share code corrections (patches) via mailing lists to combine changes coming from different developers. Git is a system which allows developers to automate such a process.

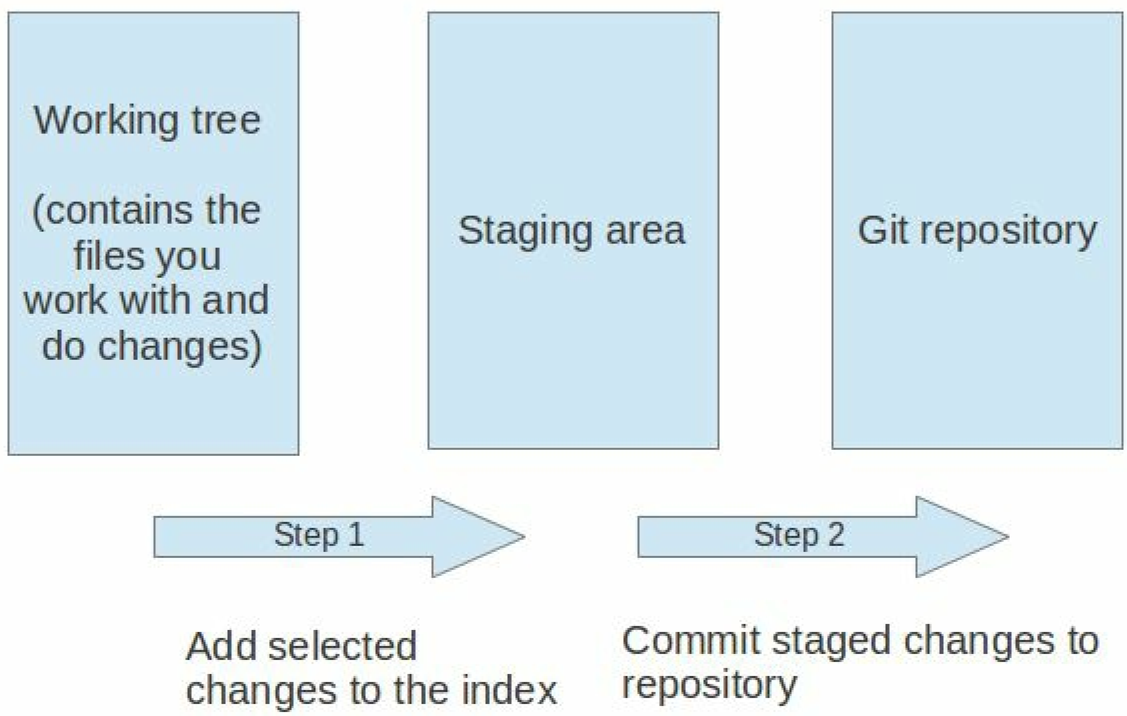
Chapter 4. The process of staging and committing

4.1. Commit process in Git

This section explains the terminology and steps involved in adding files to your Git repository.

If you modify your *working tree* (see Section 4.2, “Working tree”), e.g., by creating a new file or by changing an existing file, you need to perform two steps in Git to persist the changes in the Git repository. You first add selected files to the *staging area* and afterwards you commit the changes of the *staging area* to the Git repository.

This process is depicted in the following graphic.



4.2. Working tree

The user works on a collection of files which may originate from a certain point in time of the repository. The user may also create new files or change and delete existing ones. The current collection of files is called the working tree.

A standard Git repository contains the working tree (single checkout of one version of the project) and the full history of the repository. You can work in this *working tree* by modifying content and committing the changes to the Git repository.

4.3. Adding to the staging area

You need to mark changes in the working tree to be relevant for Git. This process is called *staging* or *to add changes to the staging area*.

You add changes in the working tree to the staging area with the `git add` command. This command stores a snapshot of the specified files in the staging area.

The `git add` command allows you to incrementally modify files, stage them, modify and stage them again until you are satisfied with your changes.

Older versions of Git used the term *index* instead of staging area. Staging area is nowadays the preferred term by the Git community. Both terms mean the same thing.

4.4. Committing to the repository

After adding the selected files to the staging area, you can *commit* these files to add them permanently to the Git repository. *Committing* creates a new persistent snapshot (called *commit* or *commit object*) of the staging area in the Git repository. A commit object, like all objects in Git, is immutable.

The *staging area* keeps track of the snapshots of the files until the staged changes are committed.

For committing the staged changes you use the `git commit` command.

4.5. Committing changes

If you commit changes to your Git repository, you create a new *commit object* in the Git repository. See Section 5.1, “Commit object (commit)” for information about the commit object./>

Chapter 5. The details of the commit objects

5.1. Commit object (commit)

Conceptually a commit object (short:commit) represents a version of all files tracked in the repository at the time the commit was created. Commits know their parent(s) and this way capture the version history of the repository.

5.2. Technical details of a commit object

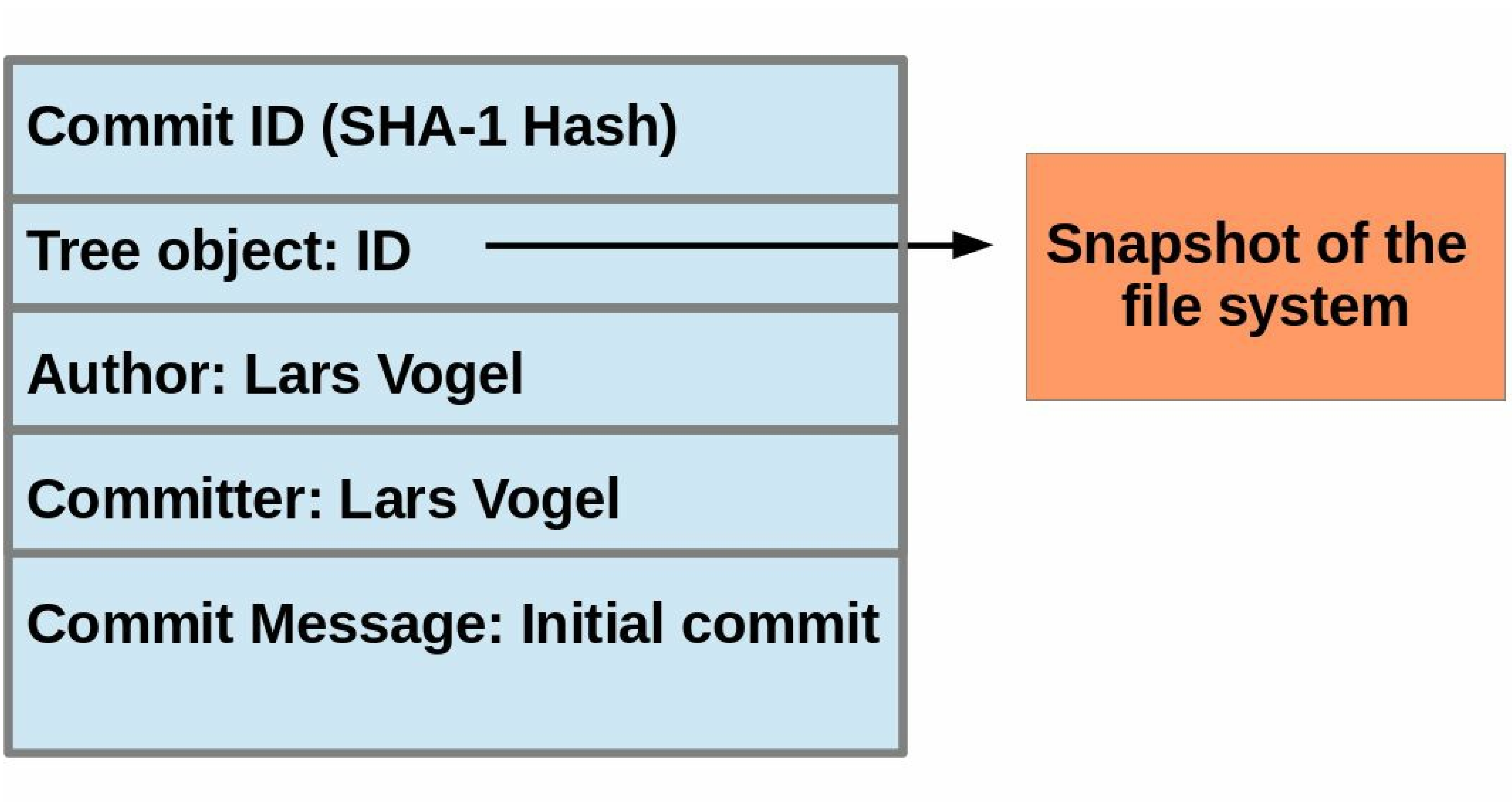
This commit object is addressable via a hash (*SHA-1 checksum*). This hash is calculated based on the content of the files, the content of the directories, the complete history of up to the new commit, the committer, the commit message, and several other factors.

This means that Git is safe, you cannot manipulate a file or the commit message in the Git repository without Git noticing that corresponding hash does not fit anymore to the content.

The *commit object* points to the individual files in this commit via a *tree* object. The files are stored in the Git repository as *blob* objects and might be packed by Git for better performance and more compact storage. Blobs are addressed via their SHA-1 hash.

Packing involves storing changes as deltas, compression and storage of many objects in a single *pack file*. *Pack files* are accompanied by one or multiple index files which speedup access to individual objects stored in these packs.

A commit object is depicted in the following picture.



The above picture is simplified. Tree objects point to other tree objects and file blobs. Objects which didn't change between commits are reused by multiple commits.

5.3. Hash and abbreviated commit hash

A Git commit object is identified by its hash (SHA-1 checksum). SHA-1 produces a 160-bit (20-byte) hash value. A SHA-1 hash value is typically rendered as a hexadecimal number, 40 digits long.

In a typical Git repository you need fewer characters to uniquely identify a commit object. As a minimum you need 4 characters and in a typical Git repository 5 or 6 are sufficient. This short form is called the abbreviated commit hash or abbreviated hash. Sometimes it is also called the shortened SHA-1 or abbreviated SHA-1.

Several commands, e.g., the `git log` command can be instructed to use the shortened SHA-1 for their output.

Part II. Terminology overview and commit references

Chapter 6. Git terminology overview

6.1. Reference table with important Git terminology

The following table provides a summary of important *Git* terminology.

Table 6.1. Important Git terminology

Term	Definition
Branch	A *branch* is a named pointer to a commit. Selecting a branch in Git terminology is called *to checkout a branch*. If you are working in a certain branch, the creation of a new commit advances this pointer to the newly created commit. Each commit knows their parents (predecessors). Successors are retrieved by traversing the commit graph starting from branches or other refs, symbolic references (for example: HEAD) orexplicit commit objects. This way a branch defines its own line of descendants in the overall version graph formed by all commits in the repository. You can create a new branch from an existing one and change the code independently from other branches. One of the branches is the default (typically named *master*). The default branch is the one for which a local branch is automatically created when cloning the repository.
Commit	When you commit your changes into a repository this creates a new *commit object* in the Git repository. This *commit object* uniquely identifies a new revision of the content of the repository. This revision can be retrieved later, for example, if you want to see the source code of an older version. Each commit object contains the author and the committer, thus making it possible to identify who did the change. The author and committer might be different people. The author did the change and the committer applied the change to the Git repository. This is common for contributions to open source projects.
HEAD	*HEAD* is a symbolic reference most often pointing to the currently checked out branch. Sometimes the *HEAD* points directly to a commit object, this is called *detached HEAD mode*. In that state creation of a commit will not move any branch. If you switch branches, the *HEAD* pointer points to the branch pointer which in turn points to a commit. If you checkout a specific commit, the *HEAD* points to this commit directly.
Index	*Index* is an alternative term for the *staging area*.
	A *repository* contains the history, the different versions over time and all different branches and tags. In Git each copy of the repository is a complete repository. If the repository is not a bare repository, it allows you to checkout revisions into your working

Repository	tree and to capture changes by creating new commits. Bare repositories are only changed by transporting changes from other repositories. This book uses the term *repository* to talk about a non-bare repository. If it talks about a bare repository, this is explicitly mentioned.
Revision	Represents a version of the source code. Git implements revisions as *commit objects* (or short *commits*). These are identified by an SHA-1 hash.
Staging area	The *staging area* is the place to store changes in the working tree before the commit. The *staging area* contains a snapshot of the changes in the working tree (changed or new files) relevant to create the next commit and stores their mode (file type, executable bit).
Tag	A *tag* points to a commit which uniquely identifies a version of the Git repository. With a tag, you can have a named point to which you can always revert to. You can revert to any point in a Git repository, but tags make it easier. The benefit of tags is to mark the repository for a specific reason, e.g., with a release. Branches and tags are named pointers, the difference is that branches move when a new commit is created while tags always point to the same commit. Tags can have a timestamp and a message associated with them.
URL	A URL in Git determines the location of the repository. Git distinguishes between *fetchurl* for getting new data from other repositories and *pushurl* for pushing data to another repository.
Working tree	The *working tree* contains the set of working files for the repository. You can modify the content and commit the changes as new commits to the repository.

6.2. File states in the working tree

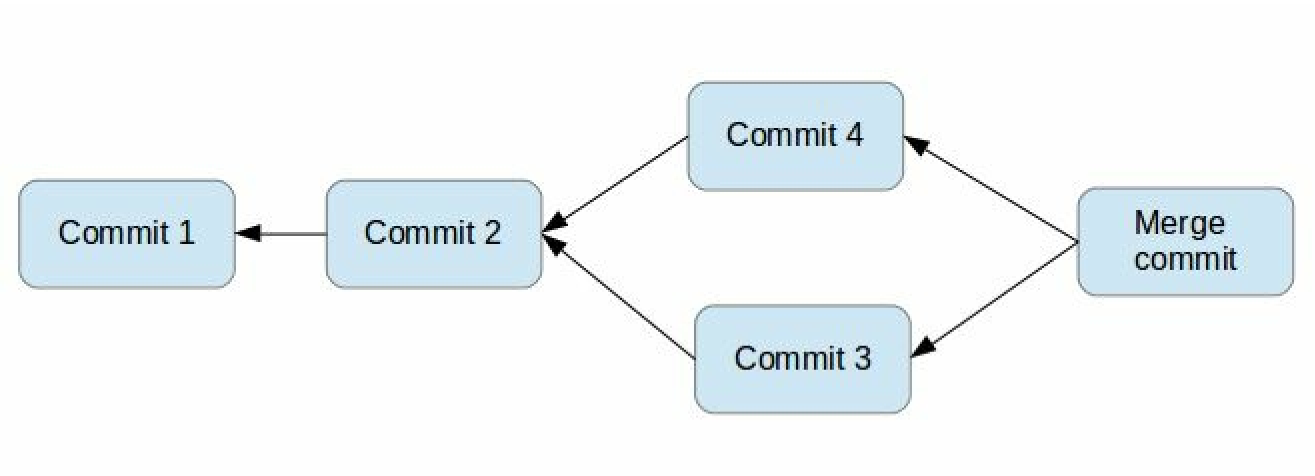
A file in the working tree of a Git repository can have different states. These states are the following:

- untracked: the file is not tracked by the Git repository. This means that the file never staged nor committed.
- tracked: committed and not staged
- staged: staged to be included in the next commit
- dirty / modified: the file has changed but the change is not staged

Chapter 7. Commit references

7.1. Predecessor commits, parents and commit references

Each commit has zero or more direct predecessor commits. The first commit has zero parents, merge commits have two or more parents, most commits have one parent.



In Git you typically need to address certain commits. For example you want to tell Git to show you all changes which were done in the last three commits. Or you want to see the differences introduced between two different branches.

Git allows addressing commits via *commit reference* for this purpose.

A commit reference can be a *simple reference* (simple ref), in this case it points directly to a commit. This is the case for a commit hash or a tag. A commit reference can also be *symbolic reference* (symbolic ref, symref). In this case it points to another reference (either simple or symbolic). For example HEAD is a symbolic ref for abranch, if it points to a branch. HEAD points to the branch pointer and the branch pointer points to a commit.

7.2. Branch references and the HEAD reference

A branch points to a specific commit. You can use the branch name as reference to the corresponding commit. You can also use HEAD to reference the corresponding commit.

7.3. Parent and ancestor commits

You can use ^ (caret) and ~ (tilde) to reference predecessor commit objects from other references. You can also combine the ^ and ~ operators. See Section 7.4, “Using caret and tilde for commit references” for their usage.

The Git terminology is *parent* for ^ and *ancestor* for ~.

7.4. Using caret and tilde for commit references

[reference]~1 describes the first predecessor of the commit object accessed via [reference]. [reference]~2 is the first predecessor of the first predecessor of the [reference] commit. [reference]~3 is the first predecessor of the first predecessor of the first predecessor of the [reference] commit, etc.

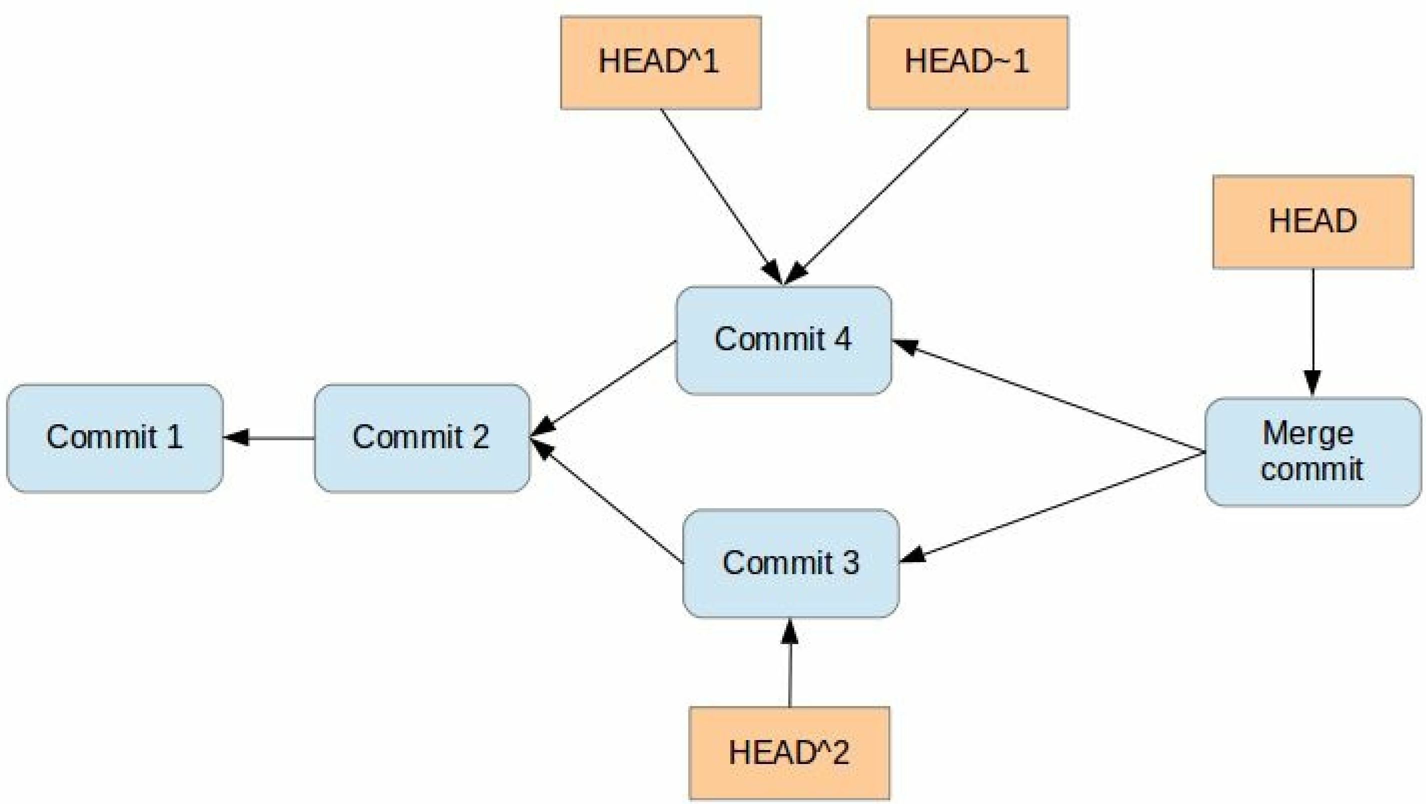
[reference]~ is an abbreviation for [reference]~1.

For example, you can use the *HEAD~1* or *HEAD~* reference to access the first parent of the commit to which the *HEAD* pointer currently points.

[reference]^1 also describes the first predecessor of the commit object accessed via [reference].

For example HEAD^^^ is the same as HEAD~~~ and is the same as HEAD~3.

The difference is that [reference]^2 describes the second parent of a commit. A merge commit typically has two predecessors. HEAD^3 means 'the third parent of a merge' and in most cases this won't exist (merges are generally between two commits, though more is possible).



[reference]^ is an abbreviation for [reference]^1.

7.5. Commit ranges with the double dot operator

You can also specify ranges of commits. This is useful for certain Git commands, for example, for seeing the changes between a series of commits.

The double dot operator allows you to select all commits which are reachable from a commit c2 but not from commit c1. The syntax for this is "c1..c2". A commit A is reachable from another commit B if A is a direct or indirect parent of B.

> **Tip**
>
> Think of c1..c2 as *all commits as of c1 (not including c1) until commit c2.*

For example, you can ask Git to show all commits which happened between HEAD and HEAD~4.

```
git log HEAD~4..HEAD
```

This also works for branches. To list all commits which are in the "master" branch but not in the "testing" branch, use the following command.

```
git log testing..master
```

You can also list all commits which are in the "testing" but not in the "master" branch.

```
git log master..testing
```

7.6. Commit ranges with the triple dot operator

The triple dot operator allows you to select all commits which are reachable either from commit c1 or commit c2 but not from both of them.

This is useful to show all commits in two branches which have not yet been combined.

```
# show all commits which
# can be reached by master or testing
# but not both
git log master...testing
```

Part III. Installation and configuration

Chapter 8. Installation of Git

8.1. Ubuntu, Debian and derived systems

On Ubuntu and similar systems you can install the Git command line tool via the following command:

```
sudo apt-get install git
```

8.2. Fedora, Red Hat and derived systems

On Fedora, Red Hat and similar systems you can install the Gitcommand line tool via the following command:

```
yum install git
```

8.3. Other Linux systems

To install Git on other Linux distributions please check the documentation of your distribution. The following listing contains the commands for the most popular ones.

```
# Arch Linux
sudo pacman -S git

# Gentoo
sudo emerge -av git

# SUSE
sudo zypper install git
```

8.4. Windows

A Windows version of Git can be found on the Git download page. This website provides native installers for each operating system.

8.5. Mac OS

The easiest way to install Git on a Mac is via the Git download page and to download and run the installer for Mac OS X.

Git is also installed by default with the Apple Developer Tools on Mac OS X.

Chapter 9. Different levels of Git configuration

9.1. Git configuration levels

The `git config` command allows you to configure your Git settings. These settings can be system wide, user or repository specific.

A more specific setting overwrites values in the previous level, i.e., a setting the repository overrides the user setting and a user setting overrides a system wide setting.

9.2. Git system-wide configuration

You can provide a system wide configuration for your Git settings. A system wide configuration is not very common, most settings are user specific or repository specific as described in the next chapters.

On a Unix based system Git uses the */etc/gitconfig* file for this system-wide configuration. To set this up, ensure you have sufficient rights, i.e. root rights, in your OS and use the `--system` option for the `git config` command.

9.3. Git user configuration

Git allows you to store user settings in the `.gitconfig` file located in the user home directory. This is also called the *global* Git configuration.

For example Git stores the committer and author of a change in each commit. This and additional information can be stored in the Git user settings.

In each Git repository you can also configure the settings for this repository. User configuration is done if you include the `--global` option in the `git config` command.

9.4. Repository specific configuration

You can also store repository specific settings in the *.git/config* file of a repository. Use the *--local* or use no flag at all. If neither the *--system* not the *--global* parameter is used, the setting is specific for the current Git repository.

Chapter 10. Configuring Git

10.1. User configuration

You have to configure at least your user and email address to be able to commit to a Git repository because this information is stored in each commit.

10.2. Exercise: User configuration

Configure your user and email for Git via the following command.

```
# configure the user which will be used by Git
# this should be not an acronym but your full name
git config --global user.name "Firstname Lastname"

# configure the email address
git config --global user.email "your.email@example.org"
```

10.3. Push configuration

If your are using Git in a version below 2.0 you should also execute the following command.

```
# set default so that only the current branch is pushed
git config --global push.default simple
```

This configures Git so that the `git push` command pushes only the active branch (in case it is connected to a remote branch, i.e., configured as remote-tracking branches) to your Git remote repository. As of Git version 2.0 this is the default and therefore it is good practice to configure this behavior.

You learn about the push command in Section 22.1, “Push changes to another repository”.

10.4. Avoid merge commits for pulling

If you pull in changes from a remote repository, Git by default creates merge commits if you pull in divergent changes. This may not be desired and you can avoid this via the following setting.

```
# set default so that you avoid unnecessary commits
git config --global branch.autosetuprebase always
```

> **Note**
>
> This setting depends on the individual workflow. Some teams prefer to create merge commits, but the author of this book likes to avoid them.

10.5. Color Highlighting

The following commands enables color highlighting for Git in the console.

```
git config --global color.ui auto
```

10.6. Setting the default editor

By default Git uses the system default editor which is taken from the *VISUAL* or *EDITOR* environment variables if set. You can configure a different one via the following setting.

```
# setup vim as default editor for Git (Linux)
git config --global core.editor vim
```

10.7. Setting the default merge tool

File conflicts might occur in Git during an operation which combines different versions of the same files. In this case the user can directly edit the file to resolve the conflict.

Git allows also to configure a merge tool for solving these conflicts. You have to use third party visual merge tools like tortoisemerge, p4merge, kdiff3 etc. A Google search for these tools help you to install them on your platform. Keep in mind that such tools are not required, you can always edit the files directly in a text editor.

Once you have installed them you can set your selected tool as default merge tool with the following command.

```
# setup kdiff3 as default merge tool (Linux)
git config --global merge.tool kdiff3

# to install it under Ubuntu use
sudo apt-get install kdiff3
```

10.8. More settings

All possible Git settings are described under the following link: git-config manual page

10.9. Query Git settings

To query your Git settings, execute the following command:

```
git config --list
```

If you want to query the global settings you can use the following command.

```
git config --global --list
```

Chapter 11. Handling line endings on different platforms

11.1. Line endings of the different platforms

Every time a developer presses return on the keyboard an invisible character called a line ending is inserted. Unfortunately, different operating systems handle line endings differently.

Linux and Mac use different line endings than Windows. Windows uses a carriage-return and a linefeed character (CRLF), while Linuxand Mac only uses a linefeed character (LF). This becomes a problem if developers use different operating system to commit changes to a Git repository.

To avoid commits because of line ending differences in your Git repository you should configure all clients to write the same line ending to the Git repository.

11.2. Configuring line ending settings as developer

On Windows systems you can tell Git to convert line endings during a checkout to CRLF and to convert them back to LF during commit. Use the following setting for this.

```
# configure Git on Windows to properly handle line endings
git config --global core.autocrlf true
```

On Linux and Mac you can tell Git to convert CRLF to LF with the following setting.

```
# configure Git on Linux and Mac to properly handle line endings
git config --global core.autocrlf input
```

11.3. Configuring line ending settings per repository

You can also configure the line ending handling per repository by adding a special `.gitattributes` file to the root folder of your Git repository. If this file is committed to the repository, it overrides the `core.autocrlf` setting of the individual developer.

In this file you can configure Git to auto detect the line endings.

> **Note**
>
> Not all graphical Git tools support the `.gitattributes` file, for example the Eclipse IDE does currently not support it. See Eclipse Bug report.

Part IV. Ignoring files and tracking empty directories

Chapter 12. Configure files and directories to ignore

12.1. Ignoring files and directories with a .gitignore file

Git can be configured to ignore certain files and directories for repository operations. This is configured via one or several `.gitignore` files. Typically, this file is located at the root of your Git repository but it can also be located in sub-directories. In the second case the defined rules are only valid for the sub-directory and below.

You can use certain wildcards in this file. `*` matches several characters. More patterns are possible and described under the following URL: [gitignore manpage]()

For example, the following `.gitignore` file tells Git to ignore the `bin` and `target` directories and all files ending with a ~.

```
# ignore all bin directories
# matches "bin" in any subfolder
bin/

# ignore all target directories
target/

# ignore all files ending with ~
*~
```

You can create the `.gitignore` file in the root directory of the working tree to make it specific for the Git repository.

> **Tip**
>
> The `.gitignore` file tells Git to ignore the specified files in Git commands. You can still add ignored files to the *staging area* of the Git repository by using the `--force` parameter, i.e. with the `git add --force [paths]` command.
>
> This is useful if you want to add, for example, auto-generated binaries, but you need to have a fine control about the version which is added and want to exclude them from the normal workflow.

It is good practice to commit the local `.gitignore` file into the Git repository so that everyone who clones this repository have it.

12.2. Global (cross-repository) .gitignore settings

You can also setup a global *.gitignore* file valid for all Git repositories via the `core.excludesfile` setting. The setup of this setting is demonstrated in the following code snippet.

```
# Create a ~/.gitignore in your user directory
cd ~/
touch .gitignore

# Exclude bin and .metadata directories
echo "bin" >> .gitignore
echo ".metadata" >> .gitignore
echo "*~" >> .gitignore
echo "target/" >> .gitignore
# for Mac
echo ".DS_Store" >> .gitignore
echo "._*" >> .gitignore

# Configure Git to use this file
# as global .gitignore

git config --global core.excludesfile ~/.gitignore
```

The global *.gitignore* file is only locally available.

12.3. Local per-repository ignore rules

You can also create local per-repository rules by editing the `.git/info/exclude` file in your repository. These rules are not committed with the repository so they are not shared with others.

This allows you to exclude, for example, locally generated files.

Chapter 13. Track empty directories

13.1. Default behaviour of Git for empty directories

Git ignores empty directories, i.e., it does not put them under version control.

13.2. Tracking empty directories

If you want to track an empty directory in your Git repository, it is a good practice to put a file called `.gitignore` in the directory. As the directory now contains a file, Git includes it into its version control mechanism.

> **Note**
>
> The file could be called anything. Others sources recommend to call the file `.gitkeep`. One problem with this approach is that `.gitkeep` is unlikely to be ignored by build systems, resulting in the `.gitkeep` file being copied to the output repository.

Part V. Exercise: A simple local Git workflow

Chapter 14. Creating a local Git repository

14.1. Target of this chapter

In this chapter you create a local Git repository. The comments (marked with #) before the commands explain the specific actions.

Open a command shell for the operations.

14.2. Create a directory

The following commands create an empty directory which is used later in this exercise to contain the working tree and the Git repository.

```
# switch to home
cd

# create a directory and switch into it
mkdir repo01
cd repo01

# create a new directory
mkdir datafiles
```

14.3. Create a new Git repository

The following explanation is based on a non-bare repository. See Section 6.1, “Reference table with important Git terminology” for the difference between a bare repository and a non-bare repository with a *working tree*.

Every Git repository is stored in the `.git` folder of the directory in which the Git repository has been created. This directory contains the complete history of the repository. The `.git/config` file contains the configuration for the repository.

The following command creates a Git repository in the current directory.

```
# you should still be in the repo01 directory
cd ~/repo01

# initialize the Git repository
# for the current directory
git init
```

All files inside the repository folder excluding the `.git` folder are the *working tree* for a Git repository.

Chapter 15. Adding files to the Git repository

15.1. Target of this chapter

In this chapter you create several files and place them under version control.

15.2. Create new content

Use the following commands to create several new files.

```
# switch to your Git repository
cd ~/repo01

# create an empty file in a new directory
touch datafiles/data.txt

# create a few files with content
ls > test01
echo "bar" > test02
echo "foo" > test03
```

15.3. See the current status of your repository

The `git status` command shows the working tree status, i.e. which files have changed, which are staged and which are not part of the staging area. It also shows which files have conflicts and gives an indication what the user can do with these changes, e.g., add them to the staging area or remove them, etc.

Run it via the following command.

```
git status
```

The output looks similar to the following listing.

```
On branch master

Initial commit

Untracked files:
  (use "git add <file>..." to include in what will be committed)

    datafiles/
    test01
    test02
    test03

nothing added to commit but untracked files present (use "git add" to track)
```

15.4. Add files to the staging area

Before committing changes to a Git repository you need to mark the changes that should be committed. This is done by adding the new and changed files to the staging area. This creates a snapshot of the affected files.

```
# add all files to the index of the Git repository
git add .
```

Afterwards run the `git status` command again to see the current status.

```
On branch master

Initial commit

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

    new file:   datafiles/data.txt
    new file:   test01
    new file:   test02
    new file:   test03
```

15.5. Change files that are staged

In case you change one of the staged files before committing, you need to add it again to the staging area to commit the new changes. This is because Git creates a snapshot of these staged files. All new changes must again be staged.

```
# append a string to the test03 file
echo "foo2" >> test03

# see the result
git status
```

Validate that the new changes are not yet staged.

```
On branch master

Initial commit

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

    new file:   datafiles/data.txt
    new file:   test01
    new file:   test02
    new file:   test03

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:   test03
```

Add the new changes to the staging area.

```
# add all files to the index of the Git repository
git add .
```

Use the `git status` command again to see that all changes are staged.

```
On branch master

Initial commit

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

    new file:   datafiles/data.txt
    new file:   test01
    new file:   test02
    new file:   test03
```

15.6. Commit staged changes to the repository

After adding the files to the Git staging area, you can commit them to the Git repository. This creates a new commit object with the staged changes in the Git repository and the HEAD reference points to the new commit. The *-m* parameter allows you to specify the commit message. If you leave this parameter out, your default editor is started and you can enter the message in the editor.

```
# commit your file to the local repository
git commit -m "Initial commit"
```

Chapter 16. Reviewing log and the Git repository

16.1. Using git log

The Git operations you performed have created a local Gitrepository in the `.git` folder and added all files to this repository via one commit. Run the `git log` command (See Section 30.1, “Using git log” for details).

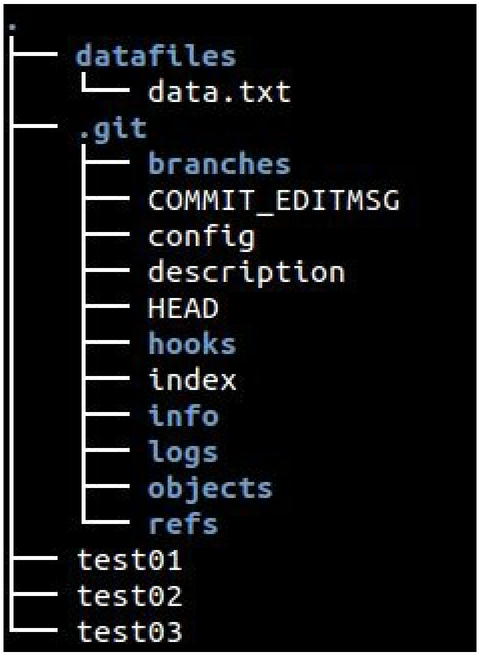
```
# show the Git log for the change
git log
```

You see an output similar to the following.

```
commit 30605803fcbd507df36a3108945e02908c823828
Author: Lars Vogel <Lars.Vogel@vogella.com>
Date:   Mon Dec 1 10:43:42 2014 +0100

    Initial commit
```

16.2. Directory structure

Your directory contains the Git repository as well as the Git working tree for your files. This directory structure is depicted in the following screenshot.



Chapter 17. Remove files and amend the last commit

17.1. Remove files

If you delete a file you use the `git add .` command to add the deletion of a file to the staging area. This is supported as of Git version 2.0.

```
# remove the "test03" file
rm test03
# add and commit the removal
git add .
# if you use Git version < 2.0 use: git add -A .
git commit -m "Removes the test03 file"
```

Alternatively you can use the `git rm` command to delete the file from your working tree and record the deletion of the file in the staging area.

17.2. Revert changes in files in the working tree

Use the `git checkout` command to reset a tracked file (a file that was once staged or committed) to its latest staged or commit state. The command removes the changes of the file in the working tree. This command cannot be applied to files which are not yet staged or committed.

```
echo "useless data" >> test02
echo "another unwanted file" >> unwantedfile.txt

# see the status
git status

# remove unwanted changes from the working tree
# CAREFUL this deletes the local changes in the tracked file
git checkout test02

# unwantedstaged.txt is not tracked by Git simply delete it
rm unwantedfile.txt
```

If you use `git status` command to see that there are no changes left in the working directory.

```
On branch master
nothing to commit, working directory clean
```

> **Warning**
>
> Use this command carefully. The `git checkout` command deletes the unstaged and uncommitted changes of tracked files in the working tree and it is not possible to restore this deletion via Git.

17.3. Correct the last commit with git amend

The `git commit --amend` command makes it possible to replace the last commit. This allows you to change the last commit including the commit message.

> **Note**
>
> The amended commit is still available until a clean-up job removes it. See Section 41.2, “git reflog” for details.

Assume the last commit message was incorrect as it contained a typo. The following command corrects this via the `--amend` parameter.

```
# assuming you have something to commit
git commit -m "message with a tpyo here"

# amend the last commit
git commit --amend -m "More changes - now correct"
```

You should use the `git --amend` command only for commits which have not been pushed to a public branch of another Git repository. The `git --amend` command creates a new commit ID and people may have based their work already on the existing commit. In this case they would need to migrate their work based on the new commit.

Chapter 18. Ignoring certain files and directories

18.1. Ignore files and directories with the .gitignore file

Git allows you to define pattern for files which should not be tracked by the Git repository. Create the following *.gitignore* file in the root of your Git directory to ignore the specified directory and file.

```
cd ~/repo01
touch .gitignore
echo ".metadata/" >> .gitignore
echo "doNotTrackFile.txt" >> .gitignore
```

> **Tip**
>
> The above command creates the file via the command line. A more common approach is to use your favorite text editor to create the file. This editor must save the file as plain text, e.g., gedit under Ubuntu or Notepad under Windows.

The resulting file looks like the following listing.

```
.metadata/
doNotTrackFile.txt
```

18.2. Stop tracking files based on the .gitignore file

Files that are tracked by Git are not automatically removed if you add them to a *.gitignore* file. Git never ignores files which are already tracked, so changes in the *.gitignore* file only affect new files. If you want to ignore files which are already tracked you need to explicitly remove them.

The following command demonstrates how to remove the *.metadata* directory and the *doNotTrackFile.txt* file from being tracked. This is example code, as you did not commit the corresponding files in your example, the command will not work in your Git repository.

```
# remove directory .metadata from git repo
git rm -r --cached .metadata
# remove file test.txt from repo
git rm --cached doNotTrackFile.txt
```

Adding a file to the *.gitignore* file does not remove the file from the repository history. If the file should also be removed from the history, have a look at `git filter-branch` which allows you to rewrite the commit history. See Section 52.1, “Using git filter-branch” for details.

18.3. Commit the .gitignore file

It is good practice to commit the *.gitignore* file into the Git repository. Use the following commands for this.

```
# add the .gitignore file to the staging area
git add .gitignore
# commit the change
git commit -m "Adds .gitignore file"
```

Part VI. Remote repositories

Chapter 19. What are remote repositories?

19.1. What are remotes?

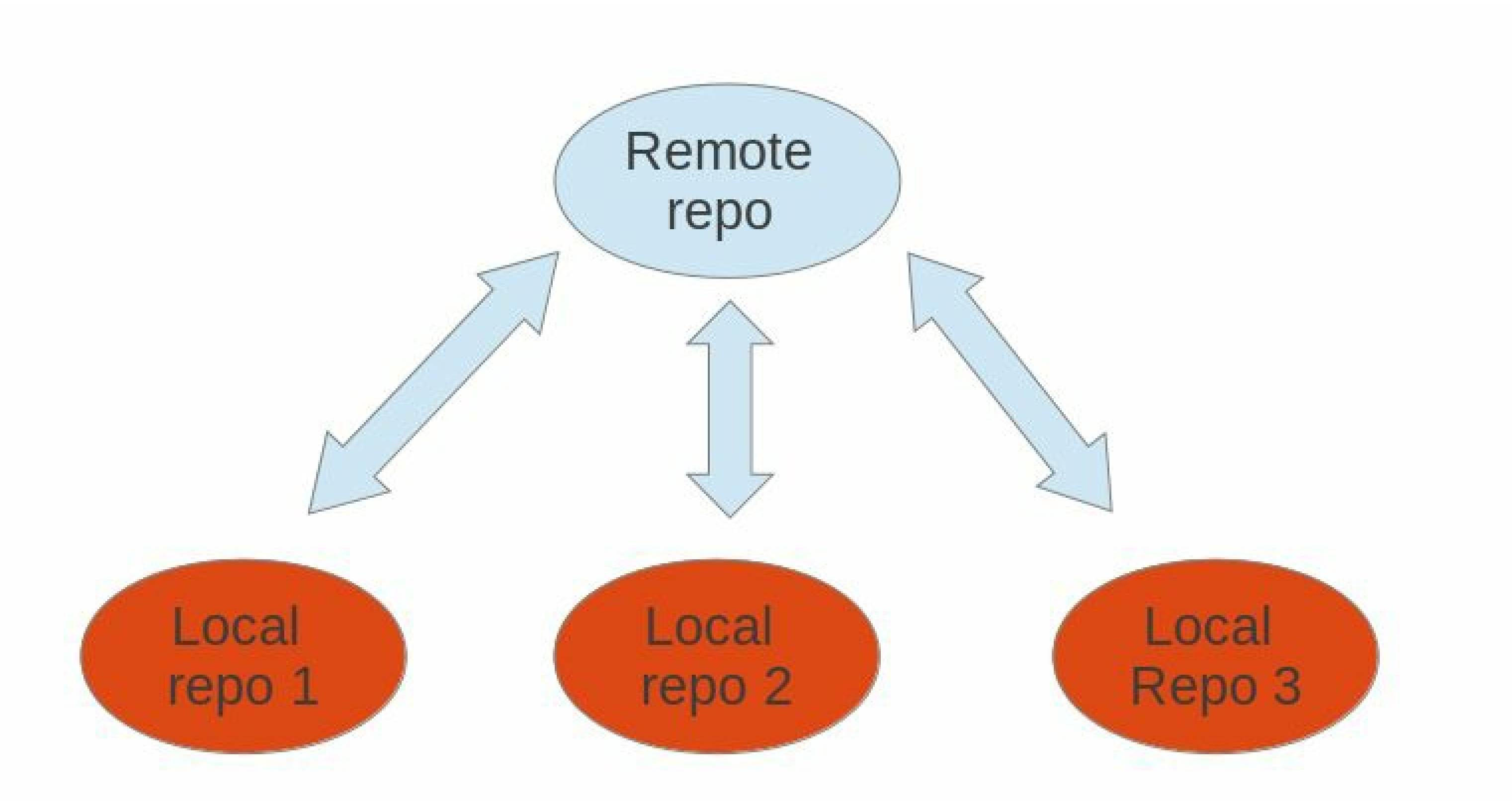
Remotes are URLs in a Git repository to otherremote repositories that are hosted on the Internet, locally or on the network.

Such remotes can be used to synchronize the changes of several Git repositories. A local Git repository can be connected to multiple remote repositories and you can synchronize your local repository with them via Git operations.

Note

Think of *remotes* as shorter bookmarks for repositories. You can always connect to a remote repository if you know its URL and if you have access to it. Without *remotes* the user would have to type the URL for each and every command which communicates with another repository.

It is possible that users connect their individual repositories directly, but a typically Git workflow involves one or more remote repositories which are used to synchronize the individual repository. Typically the remote repository which is used for synchronization is located on a server which is always available.



Tip

A remote repository can also be hosted in the local file system.

19.2. Bare repositories

A remote repository on a server typically does not require a *working tree*. A Git repository without a *working tree* is called a *bare repository*. You can create such a repository with the `--bare` option. The command to create a new empty bare remote repository is displayed below.

```
# create a bare repository
git init --bare
```

By convention the name of a bare repository should end with the `.git` extension.

To create a bare Git repository in the Internet you would, for example, connect to your server via the SSH protocol or you use some Git hosting platform, e.g., GitHub.com.

19.3. Convert a Git repository to a bare repository

Converting a normal Git repository to a bare repository is not directly support by Git.

You can convert it manually by moving the content of the *`.git`* folder into the root of the repository and by removing all others files from the working tree. Afterwards you need to update the Git repository configuration with the `git config core.bare true` command.

As this is officially not supported, you should prefer cloning a repository with the *`--bare`* option.

Chapter 20. Cloning repositories and the remote called "origin"

20.1. Cloning a repository

The `git clone` command copies an existing Git repository. This copy is a working Git repository with the complete history of the cloned repository. It can be used completely isolated from the original repository.

20.2. The remote called "origin"

If you clone a repository, Git implicitly creates a *remote* named *origin* by default. The *origin remote* links back to the cloned repository.

If you create a Git repository from scratch with the `git init` command, the *origin* remote is not created automatically.

20.3. Exercise: Cloning to create a bare Git repository

In this section you create a bare Git repository. In order to simplify the following examples, the Git repository is hosted locally in the filesystem and not on a server in the Internet.

Execute the following commands to create a bare repository based on your existing Git repository.

```
# switch to the first repository
cd ~/repo01

# create a new bare repository by cloning the first one
git clone --bare . ../remote-repository.git

# check the content of the git repo, it is similar
# to the .git directory in repo01
# files might be packed in the bare repository

ls ~/remote-repository.git
```

Tip

If you receive a warning similar to the following: `push.default is unset; its implicit value is changing in Git 2.0 from 'matching' to 'simple'`, see Section 10.3, “Push configuration” for the missing configuration.

Chapter 21. Adding and listing existing remotes

21.1. Adding a remote repository

You add as many *remotes* to your repository as desired. For this you use the `git remote add` command.

You created a new Git repository from scratch earlier. Use the following command to add a remote to your new bare repository using the *origin* name.

```
# add ../remote-repository.git with the name origin
git remote add origin ../remote-repository.git
```

21.2. Synchronizing with remote repositories

You can synchronize your local Git repository with remote repositories. These commands are covered in detail in later sections but the following command demonstrates how you can send changes to your remote repository.

```
# do some changes
echo "I added a remote repo" > test02

# commit
git commit -a -m "This is a test for the new remote origin"

# to push use the command:
# git push [target]
# default for [target] is origin
git push origin
```

21.3. Show the existing remotes

To see the existing definitions of the remote repositories, use the following command.

```
# show the details of the remote repo called origin
git remote show origin
```

To see the details of the *remotes*, e.g., the URL use the following command.

```
# show the existing defined remotes
git remote

# show details about the remotes
git remote -v
```

Chapter 22. The push and pull commands

22.1. Push changes to another repository

The `git push` command allows you to send data to other repositories. By default it sends data from your current branch to the same branch of the remote repository.

By default you can only push to bare repositories (repositories without working tree). Also you can only push a change to a remote repository which results in a fast-forward merge. See Section 44.2, “Fast-forward merge” to learn about fast-forward merges.

See Section 25.6, “Push changes of a branch to a remote repository” for details on pushing branches or the Git push manpage for general information.

22.2. Pull changes

The `git pull` command allows you to get the latest changes from another repository for the current branch.

The `git pull` command is actually a shortcut for `git fetch` followed by the `git merge` or the `git rebase` command depending on your configuration. In Section 10.4, “Avoid merge commits for pulling” you configured your Git repository so that `git pull` is a fetch followed by a rebase. See Section 43.1, “Fetch” for more information about the fetch command.

22.3. Exercise: Clone your bare repository

In this exercise you create a Git repository based on the the bare repository you created in Section 20.3, “Exercise: Cloning to create a bare Git repository”.

Clone a repository and checkout a working tree in a new directory via the following commands.

```
# switch to home
cd ~
# make new directory
mkdir repo02

# switch to new directory
cd ~/repo02
# clone
git clone ../remote-repository.git .
```

22.4. Exercise: Using the push command

Make some changes in your local repository and push them from your first repository to the remote repository via the following commands.

```
# make some changes in the first repository
cd ~/repo01

# make some changes in the file
echo "Hello, hello. Turn your radio on" > test01
echo "Bye, bye. Turn your radio off" > test02

# commit the changes, -a will commit changes for modified files
# but will not add automatically new files
git commit -a -m "Some changes"

# push the changes
git push ../remote-repository.git
```

22.5. Exercise: Using the pull command

To test the `git pull` in your example Git repositories, switch to your second repository, pull in the recent changes from the remote repository, make some changes, push them to your remote repository via the following commands.

```
# switch to second directory
cd ~/repo02

# pull in the latest changes of your remote repository
git pull

# make changes
echo "A change" > test01

# commit the changes
git commit -a -m "A change"

# push changes to remote repository
# origin is automatically created as we cloned original from this repository
git push origin
```

You can pull in the changes in your first example repository with the following commands.

```
# switch to the first repository and pull in the changes
cd ~/repo01

git pull ../remote-repository.git/

# check the changes
git status
```

Chapter 23. Working with remote repositories

23.1. Cloning remote repositories

Git supports several transport protocols to connect to other Git repositories; the native protocol for Git is also called `git`.

The following command clones an existing repository using the Git protocol. The Git protocol uses the port 9148 which might be blocked by firewalls.

```
# switch to a new directory
mkdir ~/online
cd ~/online

# clone online repository
git clone git://github.com/vogella/gitbook.git
```

If you have SSH access to a Git repository, you can also use the `ssh` protocol. The name preceding @ is the user name used for the SSH connection.

```
# clone online repository
git clone ssh://git@github.com/vogella/gitbook.git

# older syntax
git clone git@github.com:vogella/gitbook.git
```

Alternatively you could clone the same repository via the `http` protocol.

```
# the following will clone via HTTP
git clone http://github.com/vogella/gitbook.git
```

23.2. Add more remote repositories

As discussed earlier cloning repository creates a *remote* called `origin` pointing to the remote repository which you cloned from. You can push changes to this repository via `git push` as Git uses *`origin`* as default. Of course, pushing to a remote repository requires write access to this repository.

You can add more *remotes* via the `git remote add [name] [URL_to_Git_repo]` command. For example, if you cloned the repository from above via the Git protocol, you could add a new remote with the name *github_http* for the http protocol via the following command.

```
# add the HTTPS protocol
git remote add github_http https://vogella@github.com/vogella/gitbook.git
```

23.3. Rename remote repositories

To rename an existing remote repository use the `git remote rename` command. This is demonstrated by the following listing.

```
# rename the existing remote repository from
# github_http to github_testing
git remote rename github_http github_testing
```

23.4. Remote operations via HTTP

It is possible to use the HTTP protocol to clone Gitrepositories. This is especially helpful if your firewall blocks everything except HTTP or HTTPS.

```
git clone http://git.eclipse.org/gitroot/platform/eclipse.platform.ui.git
```

For secured SSL encrypted communication you should use the SSHor HTTPSprotocol in order to guarantee security.

23.5. Using a proxy

Git also provides support for HTTP access via a proxy server. The following Git command could, for example, clone a repository via HTTP and a proxy. You can either set the proxy variable in general for all applications or set it only for Git.

The following listing configures the proxy via environment variables.

```
# Linux and Mac
export http_proxy=http://proxy:8080
export https_proxy=https://proxy:8443

# Windows
set http_proxy http://proxy:8080
set https_proxy http://proxy:8080

git clone http://git.eclipse.org/gitroot/platform/eclipse.platform.ui.git
```

The following listing configures the proxy via Git config settings.

```
# set proxy for git globally
git config --global http.proxy http://proxy:8080
# to check the proxy settings
git config --get http.proxy
# just in case you need to you can also revoke the proxy settings
git config --global --unset http.proxy
```

> **Tip**
>
> Git is able to store different proxy configurations for different domains, see `core.gitProxy` in Git config manpage.

Part VII. Using branches

Chapter 24. Introduction to branches

24.1. What are branches?

Git allows you to create *branches*, i.e. named pointers to commits. You can work on different branches independently from each other. The default branch is most often called *master*.

A branch pointer in Git is 41 bytes large, 40 bytes of characters and an additional new line character. Therefore, the creating of branches in Git is very fast and cheap in terms of resource consumption. Git encourages the usage of branches on a regular basis.

If you decide to work on a branch, you *checkout* this branch. This means that Git populates the *working tree* with the version of the files from the commit to which the branch points and moves the *HEAD* pointer to the new branch.

As explained in Section 6.1, “Reference table with important Git terminology” *HEAD* is a symbolic reference usually pointing to the branch which is currently checked out.

24.2. Switching branches with untracked files

Untracked files (never added to the staging area) are unrelated to any branch. They exist only in the working tree and are ignored by Git until they are committed to the Git repository. This allows you to create a branch for unstaged and uncommitted changes at any point in time.

24.3. Switching branches with uncommitted changes

Similar to untracked files you can switch branches with unstaged or staged modifications which are not yet committed.

You can switch branches if the modifications do not conflict with the files from the branch.

If Git needs to modify a changed file during the checkout of a branch, the checkout fails with a "checkout conflict" error. This avoids that you lose changes in your files.

In this case the changes must be committed, reverted or stashed (see Section 33.1, “The git stash command”). You can also always create a new branch based on the current HEAD.

Chapter 25. Working with branches

25.1. List available branches

The `git branch` command lists all local branches. The currently active branch is marked with `*`.

```
# lists available branches
git branch
```

If you want to see all branches (including remote-tracking branches), use the *-a* for the `git branch` command. See Section 42.1, "Remote tracking branches" for information about remote-tracking branches.

```
# lists all branches including the remote branches
git branch -a
```

The *-v* option lists more information about the branches.

In order to list branches in a remote repository use the `git branch -r` command as demonstrated in the following example.

```
# lists branches in the remote repositories
git branch -r
```

25.2. Create new branch

You can create a new branch via the `git branch [newname]` command. This command allows to specify the starting point (commit id, tag, remote or local branch). If not specified the commit to which the HEAD reference points is used to create the branch.

```
# syntax: git branch <name> <hash>
# <hash> in the above is optional
git branch testing
```

25.3. Checkout branch

To start working in a branch you have to *checkout* the branch. If you *checkout* a branch, the HEAD pointer moves to the last commit in this branch and the files in the working tree are set to the state of this commit.

The following commands demonstrate how you switch to the branch called *testing*, perform some changes in this branch and switch back to the branch called *master*.

```
# switch to your new branch
git checkout testing

# do some changes
echo "Cool new feature in this branch" > test01
git commit -a -m "new feature"

# switch to the master branch
git checkout master

# check that the content of
# the test01 file is the old one
cat test01
```

To create a branch and to switch to it at the same time you can use the `git checkout` command with the *-b* parameter.

```
# create branch and switch to it
git checkout -b bugreport12

# creates a new branch based on the master branch
# without the last commit
git checkout -b mybranch master~1
```

25.4. Rename a branch

Renaming a branch can be done with the following command.

```
# rename branch
git branch -m [old_name] [new_name]
```

25.5. Delete a branch

To delete a branch which is not needed anymore, you can use the following command. You may get an error message that there are uncommited changes if you did the previous examples step by step. Use force delete (uppercase *-D*) to delete it anyway.

```
# delete branch testing
git branch -d testing
# force delete testing
git branch -D testing
# check if branch has been deleted
git branch
```

25.6. Push changes of a branch to a remote repository

You can push the changes in the current active branch to a remote repository by specifying the target branch. This creates the target branch in the remote repository if it does not yet exist.

```
# push current branch to a branch called "testing" to remote repository
git push origin testing

# switch to the testing branch
git checkout testing

# some changes
echo "News for you" > test01
git commit -a -m "new feature in branch"

# push all including branch
git push
```

This way you can decide which branches you want to push to other repositories and which should be local branches. You learn more about branches and remote repositories in Section 42.1, “Remote tracking branches”.

25.7. Differences between branches

To see the difference between two branches you can use the following command.

```
# shows the differences between
# current head of master and your_branch

git diff master your_branch
```

You can also use commit ranges as described in Section 7.5, "Commit ranges with the double dot operator" and Section 7.6, "Commit ranges with the triple dot operator". For example, if you compare a branch called *your_branch* with the *master* branch the following command shows the changes in *your_branch* and *master* since these branches diverged.

```
# shows the differences in your
# branch based on the common
# ancestor for both branches

git diff master...your_branch
```

See Chapter 31, *Viewing changes with git diff and git show* for more examples of the `git diff` command.

Part VIII. Using tags

Chapter 26. Introduction to tags

26.1. What are tags?

Git has the option to *tag* a commit in the repository history so that you find it easier at a later point in time. Most commonly, this is used to tag a certain version which has been released.

If you tag a commit, you create an annotated or lightweight tag.

26.2. Lightweight and annotated tags

Git supports two different types of tags, lightweight and annotated tags.

A *lightweight tag* is a pointer to a commit, without any additional information about the tag. An *annotated tag* contains additional information about the tag, e.g., the name and email of the person who created the tag, a tagging message and the date of the tagging. *Annotated tags* can also be signed and verified with *GNU Privacy Guard (GPG)*.

26.3. Naming conventions for tags

Tags are frequently used to tag the state of a release of the Git repository. In this case they are typically called *release tags*.

Convention is that release tags are labeled based on the [major].[minor].[patch] naming scheme, for example "1.0.0". Several projects also use the "v" prefix.

The idea is that the *patch* version is incremented if (only) backwards compatible bug fixes are introduced, the *minor* version is incremented if new, backwards compatible functionality is introduced to the public API and the *major* version is incremented if any backwards incompatible changes are introduced to the public API.

For the detailed discussion on naming conventions please see the following URL: Semantic versioning.

Chapter 27. Working with tags

27.1. List tags

You can list the available tags via the following command:

```
git tag
```

27.2. Search by pattern for a tag

You can use the `-l` parameter in the `git tag` command to search for a pattern in the tag.

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

27.3. Creating lightweight tags

To create a lightweight tag don't use the `-m`, `-a` or `-s` option.

The term *build* describes the conversion of your source code into another state, e.g., converting Java sources to an executable `JAR` file. Lightweight tags in Git are often used to identify the input for a build. Frequently this does not require additional information other than a build identifier or the timestamp.

```
# create lightweight tag
git tag 1.7.1

# see the tag
git show 1.7.1
```

27.4. Creating annotated tags

You can create a new annotated tag via the `git tag -a` command. An annotated tag can also be created using the *-m* parameter, which is used to specify the description of the tag. The following command tags the current active HEAD.

```
# create tag
git tag 1.6.1 -m 'Release 1.6.1'

# show the tag
git show 1.6.1
```

You can also create tags for a certain commit id.

```
git tag 1.5.1 -m 'version 1.5' [commit id]
```

27.5. Creating signed tags

You can use the option `-s` to create a signed tag. These tags are signed with *GNU Privacy Guard (GPG)* and can also be verified with GPG. For details on this please see thefollowing URL: Git tag manpage.

27.6. Checkout tags

If you want to use the code associated with the tag, use:

```
git checkout <tag_name>
```

> **Warning**
>
> If you checkout a tag, you are in the *detached head mode* and commits created in this mode are harder to find after you checkout a branch again. See Section 41.1, “Detached HEAD” for details.

27.7. Push tags

By default the `git push` command does not transfer tags to remote repositories. You explicitly have to push the tag with the following command.

```
# push a tag or branch called tagname
git push origin [tagname]

# to explicitly push a tag and not a branch
git push origin tag <tagname>

# push all tags
git push --tags
```

27.8. Delete tags

You can delete tags with the `-d` parameter. This deletes the tag from your local repository. By default Git does not push tag deletions to a remote repository, you have to trigger that explicitly.

The following commands demonstrate how to push a tag deletion.

```
# delete tag locally
git tag -d 1.7.0

# delete tag in remote repository
# called origin
git push origin :refs/tags/1.7.0
```

Part IX. Reviewing changes before a commit

Chapter 28. Listing changed files before a commit

28.1. Listing changed files

The `git status` command shows the status of the working tree, i.e., which files have changed, which are staged and which are not part of the staging area. It also shows which files have merge conflicts and gives an indication what the user can do with these changes, e.g., add them to the staging area or remove them, etc.

28.2. Example: Using git status

The following commands create some changes in your Git repository.

```
# make some changes
# assumes that the test01
# as well as test02 files exist
# and have been committed in the past
echo "This is a new change to the file" > test01
echo "and this is another new change" > test02

# create a new file
ls > newfileanalyzis.txt
```

The `git status` command shows the current status of your repository and suggests possible actions which the user can perform.

```
# see the current status of your repository
# (which files are changed / new / deleted)
git status
```

The output of the command looks like the following listing.

```
# On branch master
# Your branch is ahead of 'origin/master' by 1 commit.
#   (use "git push" to publish your local commits)
#
# 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:   test01
#  modified:   test02
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#  newfileanalyzis.txt
no changes added to commit (use "git add" and/or "git commit -a")
```

Chapter 29. Reviewing the changes in the files before a commit

29.1. See the differences in the working tree since the last commit

The `git diff` command allows seeing the changes in the working tree compared to the last commit.

29.2. Example: Using "git diff" to see the file changes in the working tree

In order to test this, make some changes to a file and check what the `git diff` command shows to you. Afterwards commit the changes to the repository.

```
# make some changes to the file
echo "This is a change" > test01
echo "and this is another change" > test02

# check the changes via the diff command
git diff

# optional you can also specify a path to filter the displayed changes
# path can be a file or directory
git diff [path]
```

29.3. See differences between staging area and last commit

To see which changes you have staged, i.e., you are going to commit with the next commit, use the following command.

```
# make some changes to the file
git diff --cached
```

Part X. Analyzing changes in the repository

Chapter 30. Analyzing the commit history with git log

30.1. Using git log

The `git log` command shows the history of your repository in the current branch, i.e., the list of commits.

```
# show the history of commits in the current branch
git log
```

30.2. Helpful parameters for git log

The `--oneline` parameter fits the output of the `git log` command in one line.

If you use the `--abbrev-commit` parameter, the `git log` command uses shorter versions of the SHA-1 identifier for a commit object but keeps the SHA-1 unique. This parameter uses 7 characters by default, but you can specify other numbers, e.g., `--abbrev-commit --abbrev=4`.

The `graph` parameter draws a text-based graphical representation of the branches and the merge history of the Git repository.

```
# uses shortend but unique SHA-1 values
# for the commit objects
git log --abbrev-commit

# show the history of commits in one line
# with a shortened version of the commit id
# --online is a shorthand for "--pretty=oneline --abbrev-commit"
git log --oneline


# show the history as graph including branches
git log --graph --oneline
```

For more options on the `git log` command see the [Git log manpage].

30.3. View the change history of a file

To see changes in a file you can use the *-p* option in the `git log` command.

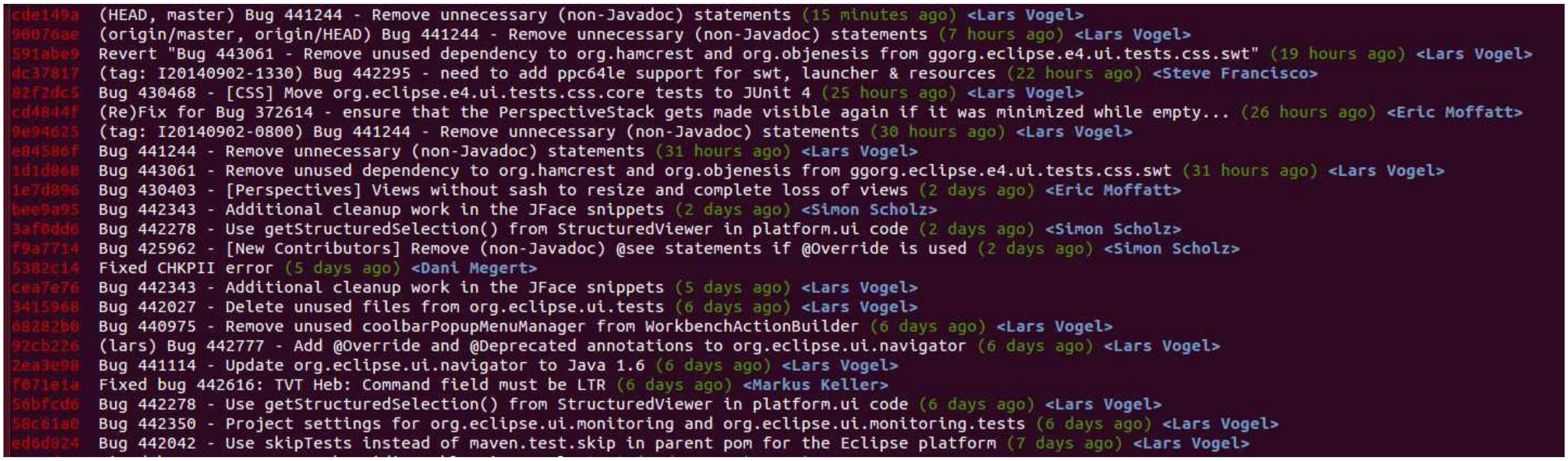
```
# git log filename shows the commits for this file
git log [file path]

# Use -p to see the diffs of each commit
git log -p filename

# --follow shows the entire history
# including renames
git log --follow -p file
```

30.4. Configuring output format

You can use the *--pretty* parameter to configure the output.

```
# command must be issued in one line, do not enter the line break
git log --pretty=format:'%Cred%h%Creset %d%Creset %s %Cgreen(%cr)
 %C(bold blue)<%an>%Creset' --abbrev-commit
```

This command creates the output.



You can define an alias for such a long command. See Section 55.1, “Using an alias” for information how to define an alias.

30.5. Searching in the commit message

You can use the `git log` command to search for a regular expression in the commit message. For example the following command instructs the log command to list all commits which contain the word "workspace" in their commit message.

```
# oneline parameter included for better readability
git log --oneline --grep="workspace"
```

30.6. See all commits of a certain user

You can use the *`--author`* option to filter the log output by a certain author. You do not need to use the full name, if a substring matches, the commit is included in the log output.

If you want to search for a specific committer use the *`--committer`* option.

The following command lists all commits with an author name containing the word "lvogel".

```
git log --author=lvogel
```

See also Section 32.3, “git shortlog for release announcements”.

Chapter 31. Viewing changes with git diff and git show

31.1. See the differences introduced by a commit

To see the changes introduced by a commit use the following command.

```
git show <commit_id>
```

31.2. See the difference between two commits

To see the differences introduced between two commits you use the `git diff` command specifying the commits. For example, the following command shows the differences introduced in the last commit.

```
# directly between two commits
git diff HEAD~1 HEAD

# using commit ranges
git diff  HEAD~1..HEAD
```

31.3. See the files changed by a commit

To see the files which have been changed in a commit use the `git diff-tree` command. The *`name-only`* tells the command to show only the names of the files.

```
git diff-tree --name-only -r <commit_id>
```

Chapter 32. Using git blame and git shortlog

32.1. Analyzing line changes with git blame

The `git blame` command allows you to see which commit and author modified a file on a per line base.

That is very useful to identify the person or the commit which introduced a change.

32.2. Example: git blame

The following code snippet demonstrates the usage of the `git blame` command.

```
# git blame shows the author and commit per
# line of a file
git blame [filename]

# the -L option allows limiting the selection
# for example by line number

# only show line 1 and 2 in git blame
git blame -L 1,2 [filename]
```

The `git blame` command can also ignore whitespace changes with the `-w` parameter.

32.3. git shortlog for release announcements

The `git shortlog` command summarizes the `git log` output, it groups all commits by author and includes the first line of the commit message.

The *-s* option suppresses the commit message and provides a commit count. The *-n* option sorts the output based on the number of commits by author.

```
# gives a summary of the changes by author
git shortlog

# compressed summary
# -s summary, provides a commit count summary only
# -n sorted by number instead of name of the author
git shortlog -sn
```

This command also allows you to see the commits done by a certain author or committer.

```
# see the commits by the author "Lars Vogel"
git shortlog --author="Lars Vogel"

# see the commits by the author "Lars Vogel"
# restricted by the last years
git shortlog --author="Lars Vogel" --since=2years

# see the number of commits by the author "Lars Vogel"
git shortlog -s --author="Lars Vogel" --since=2years
```

Part XI. Storing file changes temporary with git stash

Chapter 33. Stashing changes in Git

33.1. The git stash command

Git provides the `git stash` command which allows you to record the current state of the working directory and the staging area and to revert to the last committed revision.

This allows you to pull in the latest changes or to develop an urgent fix. Afterwards you can restore the stashed changes, which will reapply the changes to the current version of the source code.

33.2. When to use git stash

In general using the stash command should be the exception in using Git. Typically, you would create new branches for new features and switch between branches. You can also commit frequently in your local Git repository and use interactive rebase to combine these commits later before pushing them to another Git repository.

Even if you prefer not to use branches, you can avoid using the `git stash` command. In this case you commit the changes you want to put aside and amend the commit with the next commit. If you use the approach of creating a commit, you typically put a marker in the commit message to mark it as a draft, e.g., "[DRAFT] implement feature x".

Chapter 34. Using the stash command

34.1. Example: Using the git stash command

The following commands will save a stash and reapply them after some changes.

```
# create a stash with uncommitted changes
git stash

# do changes to the source, e.g., by pulling
# new changes from a remote repo

# afterwards, re-apply the stashed changes
# and delete the stash from the list of stashes
git stash pop
```

It is also possible to keep a list of stashes.

```
# create a stash with uncommitted changes
git stash save

# see the list of available stashes
git stash list
# result might be something like:
stash@{0}: WIP on master: 273e4a0 Resize issue in Dialog
stash@{1}: WIP on master: 273e4b0 Silly typo in Classname
stash@{2}: WIP on master: 273e4c0 Silly typo in Javadoc

# you can use the ID to apply a stash
git stash apply stash@{0}

# or apply the latest stash and delete it afterwards
git stash pop

# you can also remove a stashed change
# without applying it
git stash drop stash@{0}

# or delete all stashes
git stash clear
```

34.2. Create a branch from a stash

You can also create a branch for your stash if you want to continue to work on the stashed changes in a branch. This can be done with the following command.

```
# create a new branch from your stack and
# switch to it
git stash branch newbranchforstash
```

Part XII. Undoing uncommitted changes

Chapter 35. Remove untracked files with git clean

35.1. Removing untracked files

If you have untracked files in your working tree which you want to remove, you can use the `git clean` command.

> **Warning**
>
> Be careful with this command. All untracked files are removed if you run this command. You will not be able to restore them, as they are not part of your Git repository.

35.2. Example: Using git clean

The following commands demonstrate the usage of the `git clean` command.

```
# create a new file with content
echo "this is trash to be deleted" > test04

# make a dry-run to see what would happen
# -n is the same as --dry-run
git clean -n

# delete, -f is required if
# variable clean.requireForce is not set to false
git clean -f

# use -d flag to delete new directories
# use -x to delete hidden files, e.g., ".example"
git clean -fdx
```

Chapter 36. Revert uncommitted changes in tracked files

36.1. Use cases

If you have a tracked file in Git, you can always recreate the file content based on the staging area or based on a previous commit. You can also remove staged changes from the staging area to avoid that these changes are included in the next commit. This chapter explain you how you can do this.

36.2. Remove staged changes from the staging area

You can use the `git reset [paths]` command to remove staged changes from the staging area. This means that `git reset [paths]` is the opposite of `git add [paths]`. It avoids that the changes are included in the next commit. The changes are still available in the working tree, e.g., you will not loose your changes and can stage and commit them at a later point.

In the following example you create a new file and change an existing file. Both changes are staged.

```
# do changes
touch unwantedstaged.txt
echo "more.." >> test02

// add changes to staging area
git add unwantedstaged.txt
git add test02

# see the status
git status
```

The output of `git status` command should look similar to the following.

```
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    modified:   test02
    new file:   unwantedstaged.txt
```

Remove the changes from the staging area with the following command.

```
# remove test02 from the staging area
git reset test02

# remove unwantedstaged.txt from the staging area
git reset unwantedstaged.txt
```

Use the `git status` command to see the result.

```
On branch master
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:   test02

Untracked files:
  (use "git add <file>..." to include in what will be committed)

    unwantedstaged.txt

no changes added to commit (use "git add" and/or "git commit -a")
```

The `git reset` behaves differently depending on the options you provide. To learn more about the `git reset` command see Section 37.1, “Use cases for git reset”.

36.3. Remove changes in the working tree

> **Warning**
>
> Be careful with the following command. It allows you to override the changes in files in your working tree. You will not be able to restore these changes.

Changes in the working tree which are not staged can be undone with `git checkout` command. This command resets the file in the working tree to the lastest staged version. If there are no staged changes, the latest committed version is used for the restore.

```
# delete a file
rm test01

# revert the deletion
git checkout -- test01

# note git checkout test01 also works but using
# two - ensures that Git understands that test01
# is a path and not a parameter

# change a file
echo "override" > test01

# restore the file
git checkout -- test01
```

For example, you can restore the content of a directory called *`data`* with the following command.

```
git checkout -- data
```

36.4. Remove changes in the working tree and the staging area

If you want to undo a staged but uncommitted change, you use the `git checkout [commit-pointer] [paths]` command. This version of the command resets the working tree and the staged area.

The following demonstrates the usage of this to restore a delete directory.

```
# create a demo directory
mkdir checkoutheaddemo
touch checkoutheaddemo/myfile
git add .
git commit -m "Adds new directory"

# now delete the directory and add the change to
# the staging area
rm -rf checkoutheaddemo
# Use git add . -A for Git version < 2.0
git add .

# restore the working tree and reset the staging area
git checkout HEAD -- your_dir_to_restore
```

The additional commit pointer parameter instructs the `git checkout` command to reset the working tree and to also remove the staged changes.

36.5. Remove staging area based on last commit change

When you have added the changes of a file to the staging area, you can also revert the changes in the staging area base on the last commit.

```
# some nonsense change
echo "change which should be removed later" > test01

# add the file to the staging area
git add test01

# restores the file based on HEAD in the staging area
git reset HEAD test01
```

Part XIII. Undoing committed changes

Chapter 37. Resetting commited changes with git reset

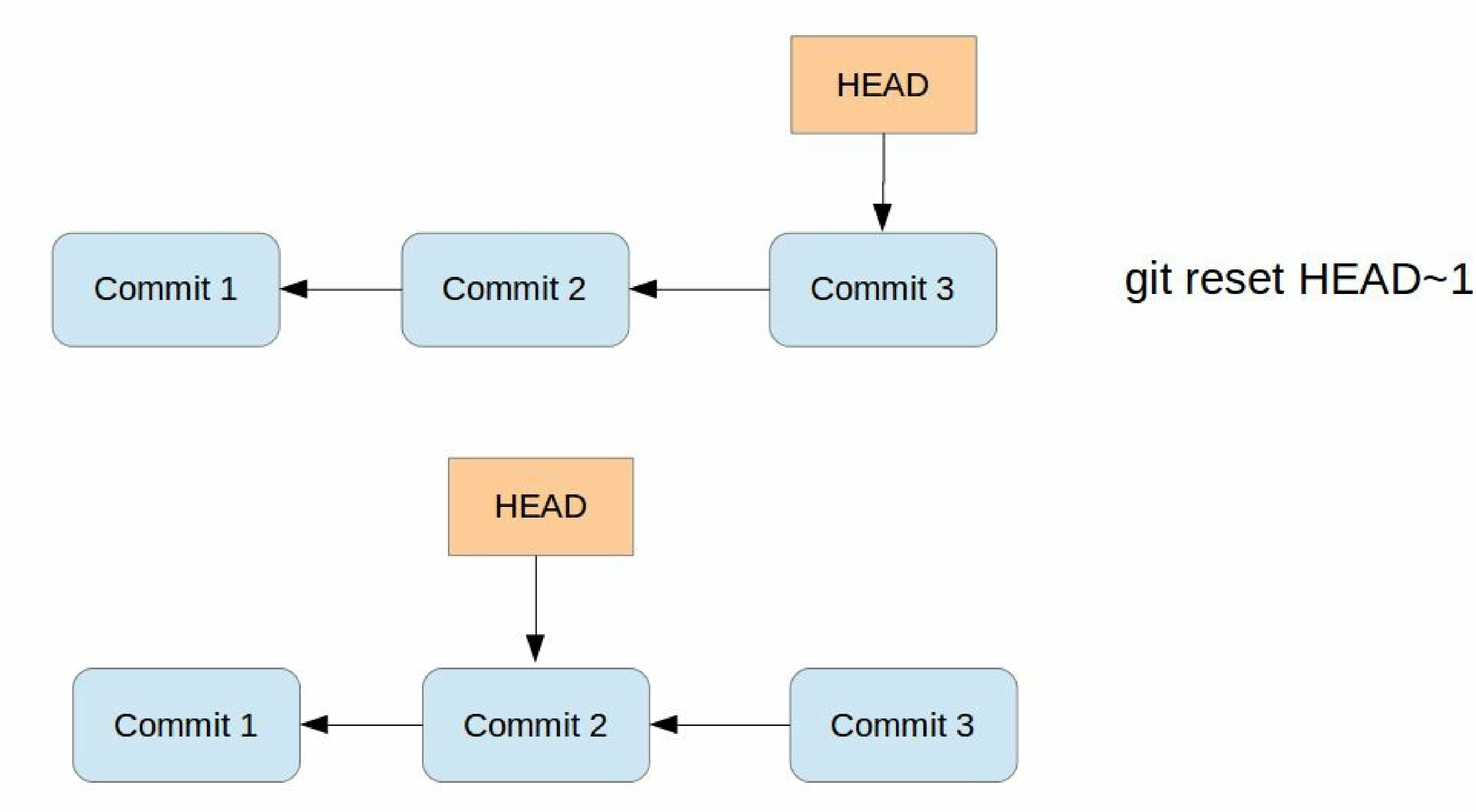
37.1. Use cases for git reset

Moving the HEAD pointer

The `git reset` command allows you to manually set the current branch to a specified state, e.g. commit. This way you can continue your work from another commit.

Depending on the specified parameters the `git reset` command performs the following:

1. If you specify the `--soft` parameter, the `git reset` command moves only the HEAD pointer.
2. If you specify the `--mixed` parameter (the default), the `git reset` command moves the HEAD pointer and resets the staging area to the new HEAD.
3. If you specify the `--hard` parameter, the `git reset` command moves the HEAD pointer and resets the staging area and the working tree to the new HEAD.



Via parameters you can define if the staging area and the working tree is updated. As a reminder, the working tree contains the files and the staging area contains the changes which are marked to be included in the next commit. These parameters are listed in the following table.

Table 37.1. git reset options

Reset	HEAD	Working tree	Staging area
soft	Yes	No	No
mixed (default)	Yes	No	Yes
hard	Yes	Yes	Yes

The `git reset` command does not remove untracked files. Use the `git clean` command for this.

Not moving the HEAD pointer with git reset

If you specify a path via the `git reset [path]` command , Git does not move the HEAD pointer. I updates the staging area or also the working tree depending on your specified option.

37.2. Finding commits that are no longer visible on a branch

If you reset the branch pointer of a branch to a certain commit, the `git log` commands does not show the commits which exist after this branch pointer. For example assume you have two commits A-> B, where B is the commit after A. You if you reset your branch pointer to A, the `git log` command does not include B anymore.

Commits like B can still be found via the `git reflog` command. See Chapter 41, *Git reflog and restoring commits*.

37.3. Deleting changes in the working tree and staging area for tracked files

The `git reset --hard` command makes the working tree exactly match HEAD.

```
# removes staged and working tree changes
# of committed files
git reset --hard
```

> **Warning**
>
> If you have tracked files with modifications, you lose these changes with the above command.

> **Note**
>
> The reset command does not delete untracked files. If you want to delete them also see Section 35.1, “Removing untracked files”.

37.4. Using git reset to squash commits

As a soft reset does not remove your change to your files and index, you can use the `git reset --soft` command to squash several commits into one commit.

As the staging area is not changed with a soft reset, you keep it in the desired state for your new commit. This means that all the file changes from the commits which were resetted are still part of the staging area.

```
# squashes the last two commits
git reset --soft HEAD~1 && git commit -m "new commit message"
```

The interactive rebase adds more flexibility to squashing commits and allows to use the existing commit messages. See Section 51.1, "Editing history with the interactive rebase" for details.

Chapter 38. Revert commits

38.1. Reverting a commit

You can revert commits via the `git revert` command. This command reverts the changes of a commit.

Such commits are useful to document that a change was withdrawn.

38.2. Example: Reverting a commit

The following command demonstrates the usage of the `git revert` command.

```
# revert a commit
git revert commit_id
```

Part XIV. Recovering files or commits

Chapter 39. Resetting the working tree based on a commit

39.1. Checkout based on commits and working tree

You can check out older revisions of your file system via the git checkout command followed by the commit ID. This command will resetyour complete *working tree* to the status described by this commit.

The commit ID is shown if you enter the `git log` command.

The following command shows the log.

```
# displays the commit history of the repository
# which contains the commit ID, author, message etc.
git log
```

The following listing shows an example output of a Git log command.

```
commit 046474a52e0ba1f1435ad285eae0d8ef19d529bf
Author: Lars Vogel <Lars.Vogel@gmail.com>
Date:   Wed Jun 5 12:13:04 2013 +0200

    Bug 409373 - Updates version number of e4 tools

    Repairs the build

commit 2645d7eef0e24195fc407137200fe7e1795ecf49
Author: Lars Vogel <Lars.Vogel@gmail.com>
Date:   Wed Jun 5 12:00:53 2013 +0200

    Bug 409373 - Updates version number of e4 CSS spy features
```

39.2. Example: Checkout a commit

To checkout a specific commit you can use the following command.

```
# checkout the older revision via
git checkout [commit_id]

# based on the example output this could be
git checkout 046474a52e0ba1f1435ad285eae0d8ef19d529bf

# or you can use the abbreviated version
git checkout 046474a5
```

Warning

If you checkout a commit, you are in the *detached head mode* and commits in this mode are harder to find after you checkout another branch. Before committing it is good practice to create a new branch to leave the *detached head mode*. See Section 41.1, “Detached HEAD” for details.

Chapter 40. Retrieving files from the history

40.1. View file in different revision

The `git show` command allows to see and retrieve files from branches, commits and tags. It allows seeing the status of these files in the selected branch, commit or tag without checking them out into your working tree.

By default, this command addresses a file from the root of the repository, not the current directory. If you want the current directory then you have to use the ./ specifier. For example to address the *pom.xml* file the current directory use: *./pom.xml*

The following commands demonstrate that. You can also make a copy of the file.

```
# [reference] can be a branch, tag, HEAD or commit ID
# [file_path] is the file name including path

git show [reference]:[file_path]

# to make a copy to copiedfile.txt

git show [reference]:[file_path] > copiedfile.txt

# assume you have two pom.xml files. One in the root of the Git
# repository and one in the current working directory

# address the pom.xml in the git root folder
git show HEAD:pom.xml

# address the pom in the current directory
git show HEAD:./pom.xml
```

40.2. Restore a deleted file in a Git repo

You can checkout a file from the commit. To find the commit which deleted the file you can use the `git log` or the `git ref-list` command as demonstrated by the following command.

```
# see history of file
git log -- <file_path>

# checkout file based on predecessors the last commit which affect it
# this was the commit which delete the file
git checkout [commit] ^ -- <file_path>

# alternatively use git rev-list
git rev-list -n 1 HEAD -- <file_path>

# afterwards, the same checkout based on the predecessors
git checkout [commit] ^ -- <file_path>
```

40.3. See which commit deleted a file

The `git log` command allows you to determine which commit deleted a file. You can use the `--` option in `git log` to see the commit history for a file, even if you have deleted the file.

```
# see the changes of a file, works even
# if the file was deleted
git log -- [file_path]

# limit the output of Git log to the
# last commit, i.e. the commit which delete the file
# -1 to see only the last commit
# use 2 to see the last 2 commits etc
git log -1 -- [file_path]

# include stat parameter to see
# some statics, e.g., how many files were
# deleted
git log -1 --stat -- [file_path]
```

Chapter 41. Git reflog and restoring commits

41.1. Detached HEAD

If you checkout a commit or a tag, you are in the so-called *detached HEAD mode*. If you commit changes in this mode, you have no branch which points to this commit. After you checkout a branch you cannot see the commit you did in detached head mode in the `git log` command.

To find such commits you can use the `git reflog` command.

41.2. git reflog

Reflog is a mechanism to record the movements of the *HEAD* and the branches references.

The Git reflog command gives a history of the complete changes of the *HEAD* reference.

```
git reflog
# <output>
# ... snip ...
1f1a73a HEAD@{2}: commit: More chaanges - typo in the commit message
45ca204 HEAD@{3}: commit: These are new changes
cf616d4 HEAD@{4}: commit (initial): Initial commit
```

The `git reflog` command also list commits which you have removed.

> **Tip**
>
> There are multiple reflogs: one per branch and one for HEAD.For branches use the `git reflog [branch]` command and for HEAD use the`git reflog` or the `git reflog HEAD` command.

41.3. Example

The following example shows how you can use git reflog to reset the current local branch to a commit which isn't reachable from the current branch anymore.

```
# assume the  ID for the second commit is
# 45ca2045be3aeda054c5418ec3c4ce63b5f269f7

# resets the head for your tree to the second commit
git reset --hard 45ca2045be3aeda054c5418ec3c4ce63b5f269f7

# see the log
git log

# output shows the history until the 45ca2045be commit

# see all the history including the deletion
git reflog

# <output>
cf616d4 HEAD@{1}: reset: moving to 45ca2045be3aeda054c5418ec3c4ce63b5f269f7
# ... snip ...
1f1a73a HEAD@{2}: commit: More chaanges - typo in the commit message
45ca204 HEAD@{3}: commit: These are new changes
cf616d4 HEAD@{4}: commit (initial): Initial commit

git reset --hard 1f1a73a
```

Part XV. Remote and local tracking branches and git fetch

Chapter 42. Local and remote-tracking branches

42.1. Remote tracking branches

Your local Git repository contains references to the state of the branches on the remote repositories to which it is connected. These local references are called *remote-tracking branches*.

You can see your remote-tracking branches with the following command.

```
# list all remote branches
git branch -r
```

To update remote-tracking branches without changing local branches you use the `git fetch` command which is covered in Chapter 43, *Git fetch*.

42.2. Delete a remote-tracking branch in your local repository

It is also safe to delete a remote branch in your local Gitrepository. You can use the following command for that.

```
# delete remote branch from origin

git branch -d -r origin/[remote_branch]
```

The next time you run the `git fetch` command the remote branch is recreated.

42.3. Delete a branch in a remote repository

To delete the branch in a remote repository use the following command.

```
# delete branch in a remote repository
git push [remote] :branch
```

Alternatively you can also use the following command.

```
# delete branch in a remote repository

git push [remote] --delete :[branch]
```

For example if you want to delete the branch called *testbranch* in the remote repository called *origin* you can use the following command.

```
git push origin :testbranch
```

> **Note**
>
> Note you can also specify the remote repository's URL. So thefollowing command also works.
>
> ```
> git push ssh://[URL_to_repo] :testbranch
> ```

42.4. Tracking branches

Branches can track another branch. This is called *to have an upstream branch* and such branches can be referred to as *tracking branches*.

Tracking branches allow you to use the `git pull` and `git push` command directly without specifying the branch and repository.

If you clone a Git repository, your local *master* branch is created as a *tracking branch* for the *master* branch of the *origin* repository (short: *origin/master*) by Git.

42.5. Setting up tracking branches

You create new *tracking branches* by specifying the *remote branch* during the creation of a branch. The following example demonstrates that.

```
# setup a tracking branch called newbrach
# which tracks origin/newbranch
git checkout -b newbranch origin/newbranch
```

Instead of using the `git checkout` command you can also use the `git branch` command.

```
# origin/master used as example, but can be replaced

# create branch based on remote branch
git branch [new_branch] origin/master

# use --track,
# default when the start point is a remote-tracking branch
git branch --track [new_branch] origin/master
```

The `--no-track` allows you to specify that you do not want to track a branch. You can explicitly add a tracking branch with the `git branch -u` command later.

```
# instruct Git to create a branch which does
# not track another branch
git branch --no-track [new_branch_notrack] origin/master

# update this branch to track the origin/master branch
git branch -u origin/master [new_branch_notrack]
```

42.6. See the branch information for a remote repository

To see the tracking branches for a remote repository (short: remote) you can use the following command.

```
# show all remote and tracking branches for origin
git remote show origin
```

An example output of this might look as follows.

```
* remote origin
  Fetch URL: ssh://test@git.eclipse.org/gitroot/e4/org.eclipse.e4.tools.git
  Push  URL: ssh://test@git.eclipse.org/gitroot/e4/org.eclipse.e4.tools.git
  HEAD branch: master
  Remote branches:
    integration                tracked
    interm_rc2                 tracked
    master                     tracked
    smcela/HandlerAddonUpdates tracked
  Local branches configured for 'git pull':
    integration rebases onto remote integration
    master      rebases onto remote master
    testing     rebases onto remote master
  Local refs configured for 'git push':
    integration pushes to integration (up to date)
    master      pushes to master      (up to date)
```

Chapter 43. Git fetch

43.1. Fetch

The `git fetch` command updates your remote-tracking branches, i.e., it updates the local copy of branches stored in a remote repository. The following command updates the remote-tracking branches from the repository called *origin*.

```
git fetch origin
```

The fetch command only updates the *remote-tracking branches* and none of the local branches and it does not change the working tree of the Git repository. Therefore, you can run the `git fetch` command at any point in time.

After reviewing the changes in the remote tracking branch you can merge the changes into your local branches or rebase your local branches onto the remote-tracking branch.

Alternatively you can also use the `git cherry-pick commit_id` command to take over only selected commits.

See Section 47.1, “Applying a single commit” for information about cherry-pick. See Section 44.1, “Merging” for the merge operation and Section 46.1, “Rebasing branches” for the rebase command.

43.2. Fetch from all remote repositories

The `git fetch` command updates only the remote-tracking branches for one remote repository. In case you want to update the remote-tracking branches of all your remote repositories you can use the following command.

```
# simplification of the fetch command
# this runs git fetch for every remote repository
git remote update

# the same but remove all stale branches which
# are not in the remote anymore
git remote update --prune
```

43.3. Compare remote-tracking branch with local branch

The following code shows a few options how you can compare your branches.

```
# show the log entries between the last local commit and the
# remote branch
git log HEAD..origin/master

# show the diff for each patch
git log -p HEAD..origin/master

# show a single diff
git diff HEAD...origin/master

# instead of using HEAD you can also
# specify the branches directly
git diff master origin/master
```

The above commands show the changes introduced in HEAD comparedto origin. If you want to see the changes in origin compared to HEAD, you can switch the arguments or use the `-R` parameter.

43.4. Rebase your local branch onto the remote-tracking branch

You can rebase your current local branch onto a remote-tracking branch. The following commands demonstrate that.

```
# assume you want to rebase master based on the latest fetch
# therefore check it out
git checkout master

# update your remote-tracking branch
git fetch

# rebase your master onto origin/master
git rebase origin/master
```

Tip

More information on the rebase command can be found in Section 46.1, “Rebasing branches”.

43.5. Fetch compared with pull

The `git pull` command performs a `git fetch` and `git merge` (or `git rebase` based on your Git settings). The `git fetch` does not perform any operations on your local branches. You can always run the fetch command and review the incoming changes.

Part XVI. Merging changes of branches

Chapter 44. Merge branches and types of merges

This chapter explains how to merge changes between two different branches under the assumption that no merging conflicts happen. Solving conflicts is covered in Section 48.1, “What is a conflict during a merge operation?”.

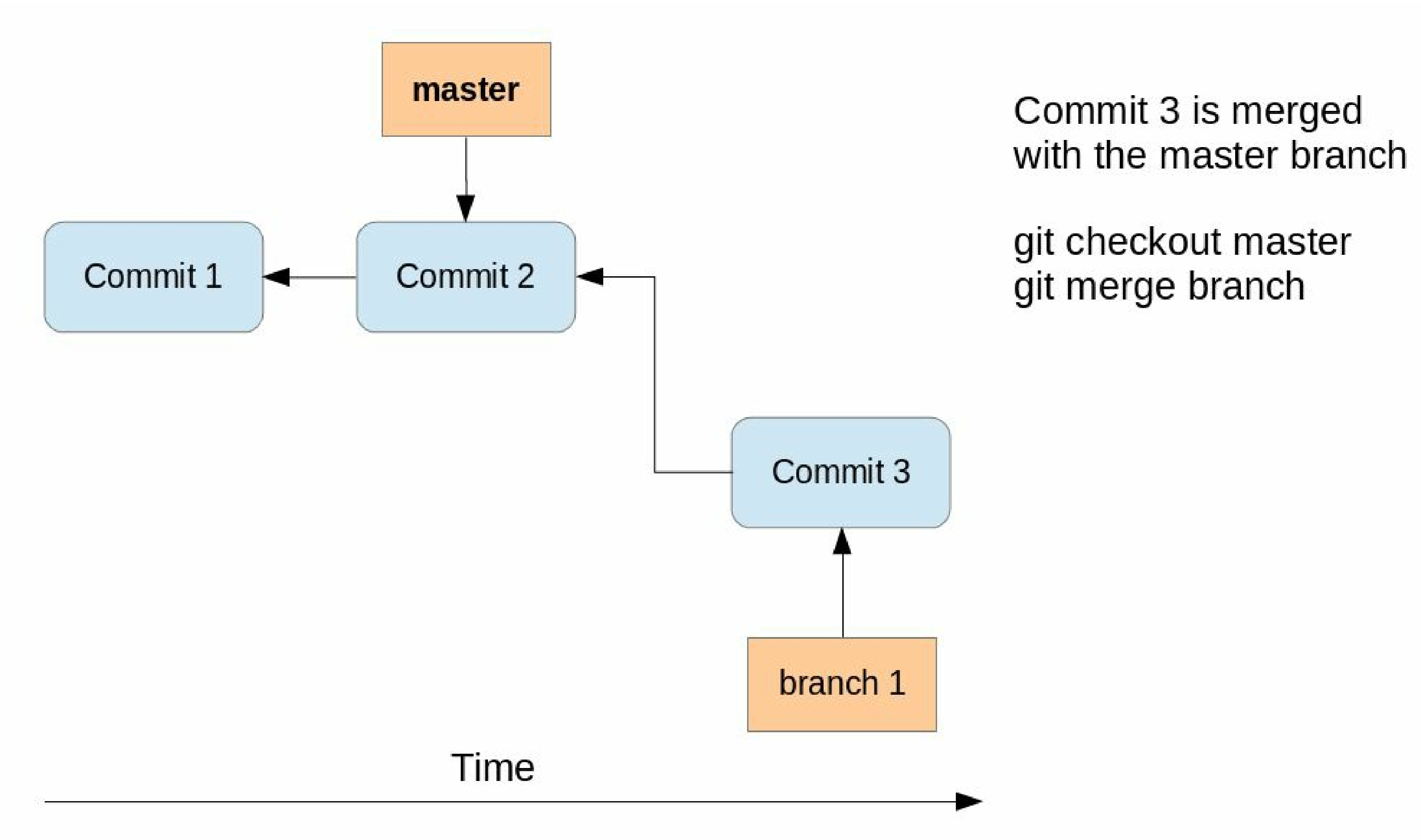
44.1. Merging

Git allows you to combine the changes which were created on two different branches. One way to achieve this is *merging*, which is described in this chapter. Other ways are using rebase or cherry-pick.

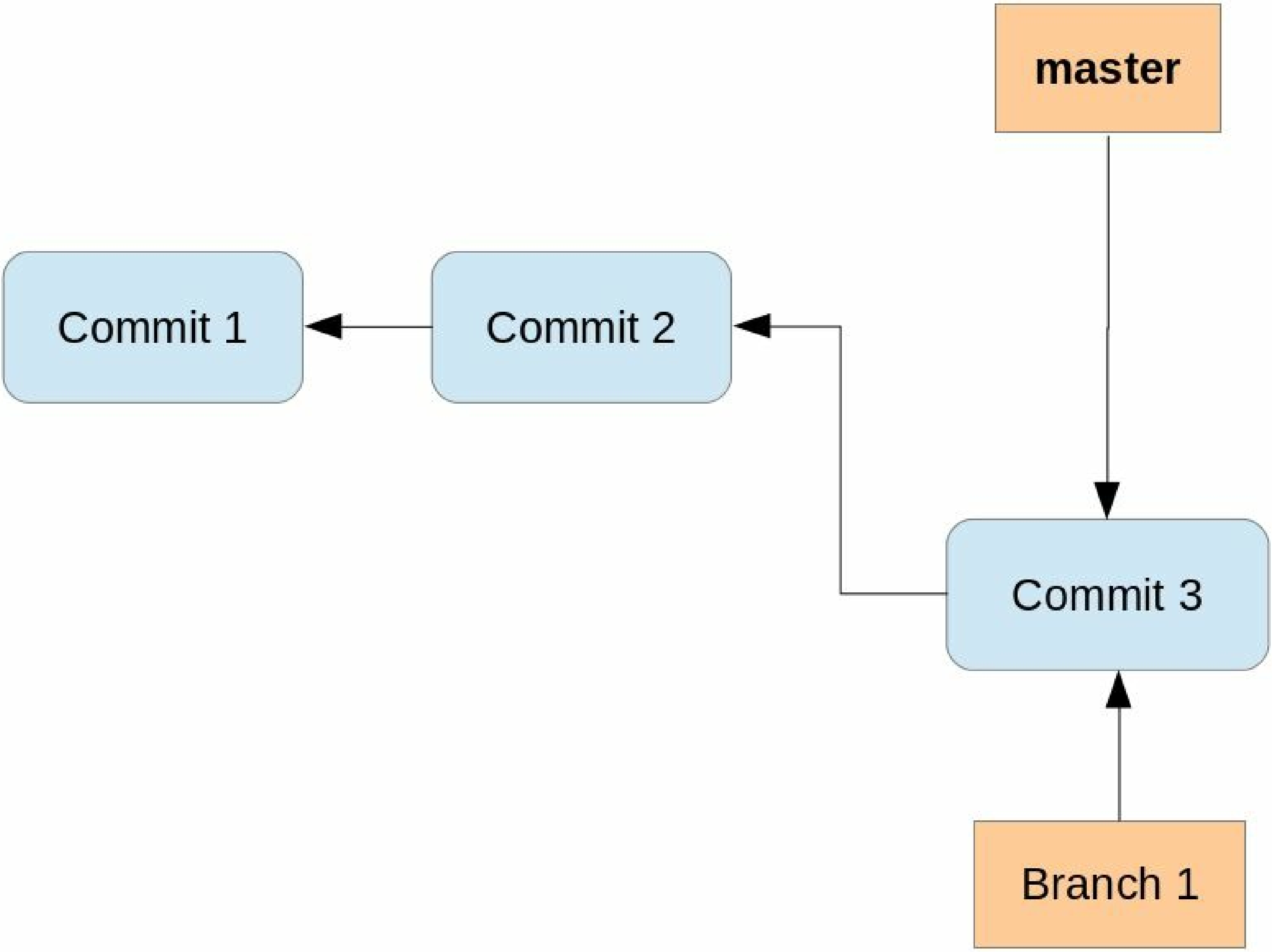
44.2. Fast-forward merge

If the commits which are merged are direct successors of the *HEAD* pointer of the current branch, Git simplifies things by performing a so-called *fast forward merge*. This *fast forward merge* simply moves the *HEAD* pointer of the current branch to the tip of the branch which is being merged. You can also merge based on a tag or a commit.

This process is depicted in the following diagram. The first picture assumes that master is checked out and that you want to merge the changes of the branch labeled "branch 1" into your "master" branch. Each commit points to its predecessor (parent).

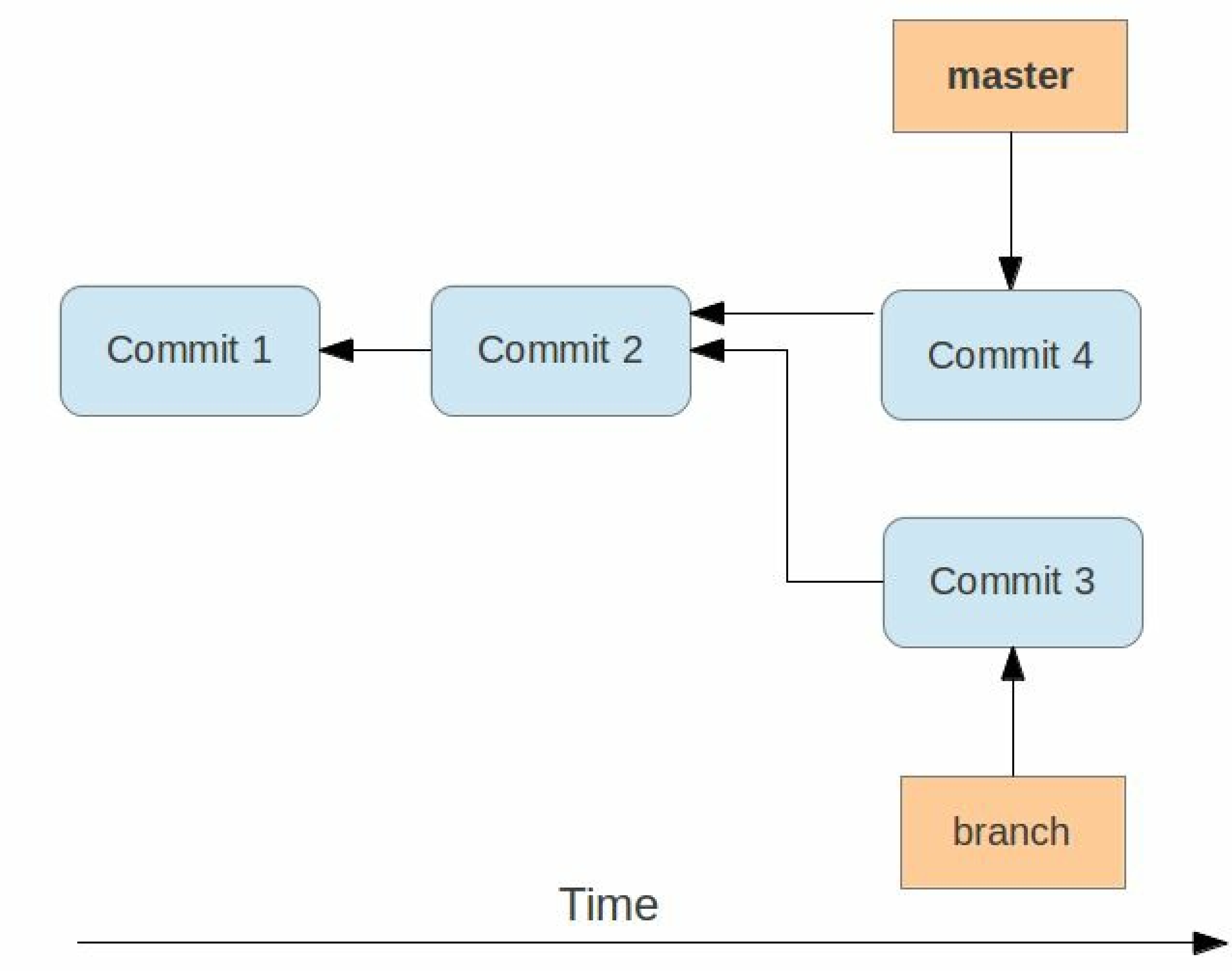


After the fast-forward merge the *HEAD* points to the master branch pointing to "Commit 3". The "branch 1" branch points to the same commit.

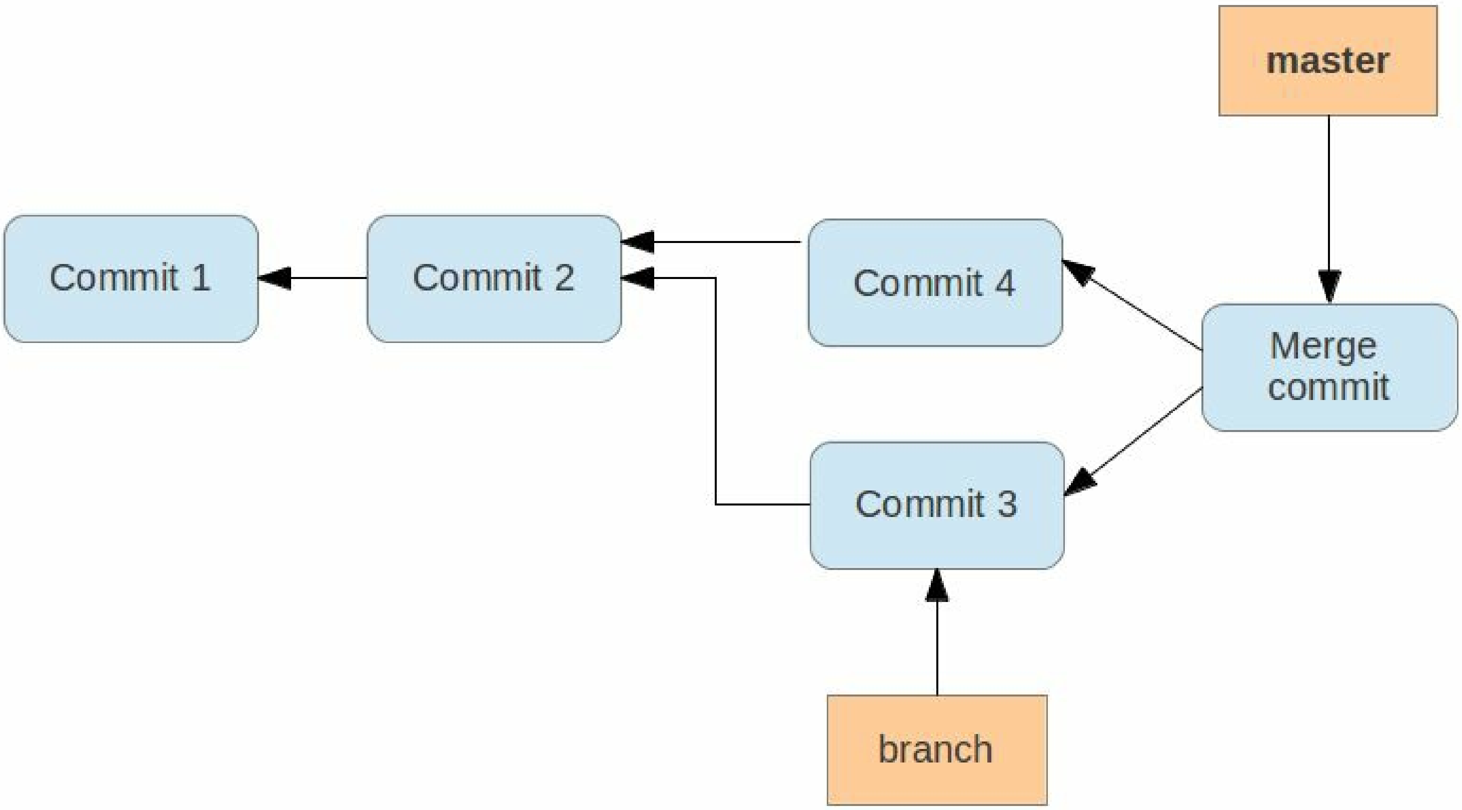

master
Commit 1
Commit 2
Commit 3
Branch 1

44.3. Merge commit

If commits are merged which are not direct predecessors of the current branch, Git performs a so-called *three-way-merge* between the latest commits of the two branches, based on the most recent common predecessor of both.



As a result a so-called *merge commit* is created on the current branch which combines the respective changes from the two branches being merged. This commit points to both of its predecessors.



Note

If multiple common predecessors exist, Git uses recursion to create a virtual common predecessor. For this Git creates a merged tree of the common ancestors and uses that as the reference for the 3-way merge. This is called the *recursive merge* strategy and is the default merge strategy.

44.4. Merge strategies - Octopus, Subtree, Ours

If a fast-forward merge is not possible, Git uses a merge strategy. The default strategy called *recursive merge* strategy was described in Section 44.3, "Merge commit".

The Git command line tooling also supports the *octopus merge* strategy for merges of multiple references. With this operation it can merge multiple branches at once.

The `subtree` option is useful when you want to merge in another project into a sub-directory of your current project. It is rarely used and you should prefer the usage of Git submodules. See Section 53.1, "What are submodules?" for more information.

The `ours` strategy merges a branch without looking at the changes introduced in this branch. This keeps the history of the merged branch but ignores the changes introduced in this branch.

You typically use the *ours* merge strategy to document in the Git repository that you have integrated a branch and decided to ignore all changes from this branch.

See also Section 45.3, "Specifying parameters for the default merge strategy" for a discussion how to use merge strategies and additional parameters.

Chapter 45. Performing a merge operation

45.1. The git merge command

The `git merge` command performs a merge. You can merge changes from one branch to the current active one via the following command.

```
# syntax: git merge <branch-name>
# merges into your currently checked out branch
git merge testing
```

45.2. Specifying merge strategies

The `-s` parameter allows you to specify other merge strategies. This is demonstrated with the following command.

For example, you can specify the *ours* strategy in which the result of the merge is always that of the current branch head, effectively ignoring all changes from all other branches. This is demonstrated with the following command.

```
# merge branch "obsolete" ignoring all
# changes in the branch
git merge -s ours obsolete
```

> **Warning**
>
> Be careful if you use the *ours* merge strategy, it ignores everything from the branch which is merged.

The usage of the octopus merge strategy is triggered if you specify more than one reference to merge.

```
# merge the branch1 and the branch2 using
# changes in the branch
git merge branch1 branch2</code>
```

45.3. Specifying parameters for the default merge strategy

The recursive merge strategy (default) allows you to specify flags with the `-X` parameter. For example you can specify here the *ours* option. This option forces conflicting changes to be auto-resolved by favoring the local version. Changes from the other branch that do not conflict with our local version are reflected to the merge result. For a binary file, the entire contents are taken from the local version.

> **Warning**
>
> The *ours* option for the *recursive* merge strategy should not be confused with the *ours* merge strategy.

A similar option to *ours* is the *theirs* option. This option prefers the version from the branch which is merged.

Both options are demonstrated in the following example code.

```
# merge changes preferring our version
git merge -s recursive -X ours [branch_to_merge]

# merge changes preferring the version from
# the branch to merge
git merge -s recursive -X theirs [branch_to_merge]
```

Another useful option is the *ignore-space-change* parameter which ignores whitespace changes.

For more information about the merge strategies and options see Git merge manpage.

45.4. Enforcing the creation of a merge commit

If you prefer to have merge commits even for situations in which Git could perform a fast-forward merge you can use the `git merge --no-ff` command.

The *`--no-ff`* parameter can make sense if you want to record in the history at which time you merged from a maintenance branch to the master branch.

When pulling from a remote repository, prefer doing a rebase to a merge. This will help to keep the history easier to read. A merge commit can be helpful to document that functionality was developed in parallel.

Part XVII. Rebase and cherry-pick

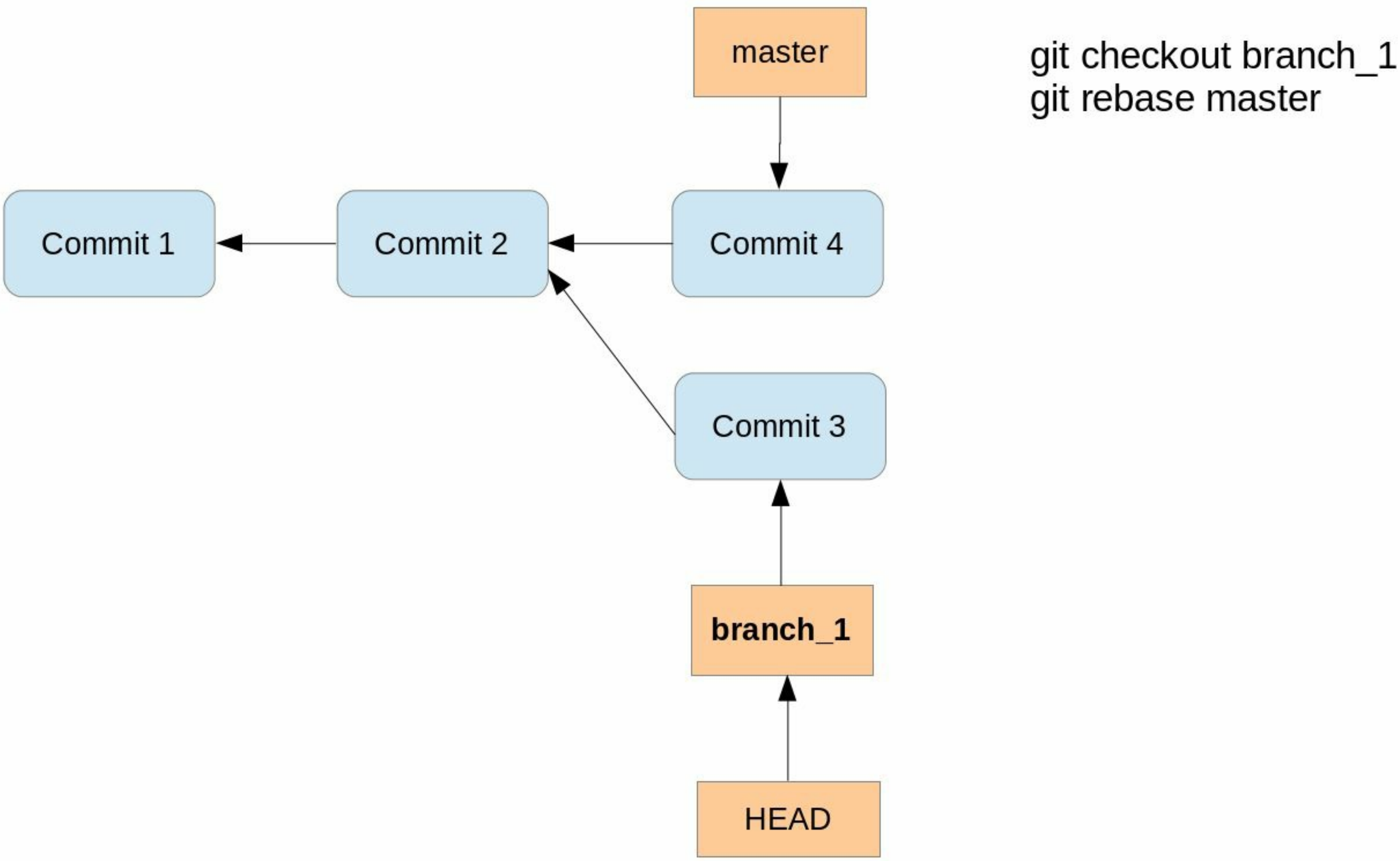
Chapter 46. Rebase branches

This chapter explains the rebase operation in Git

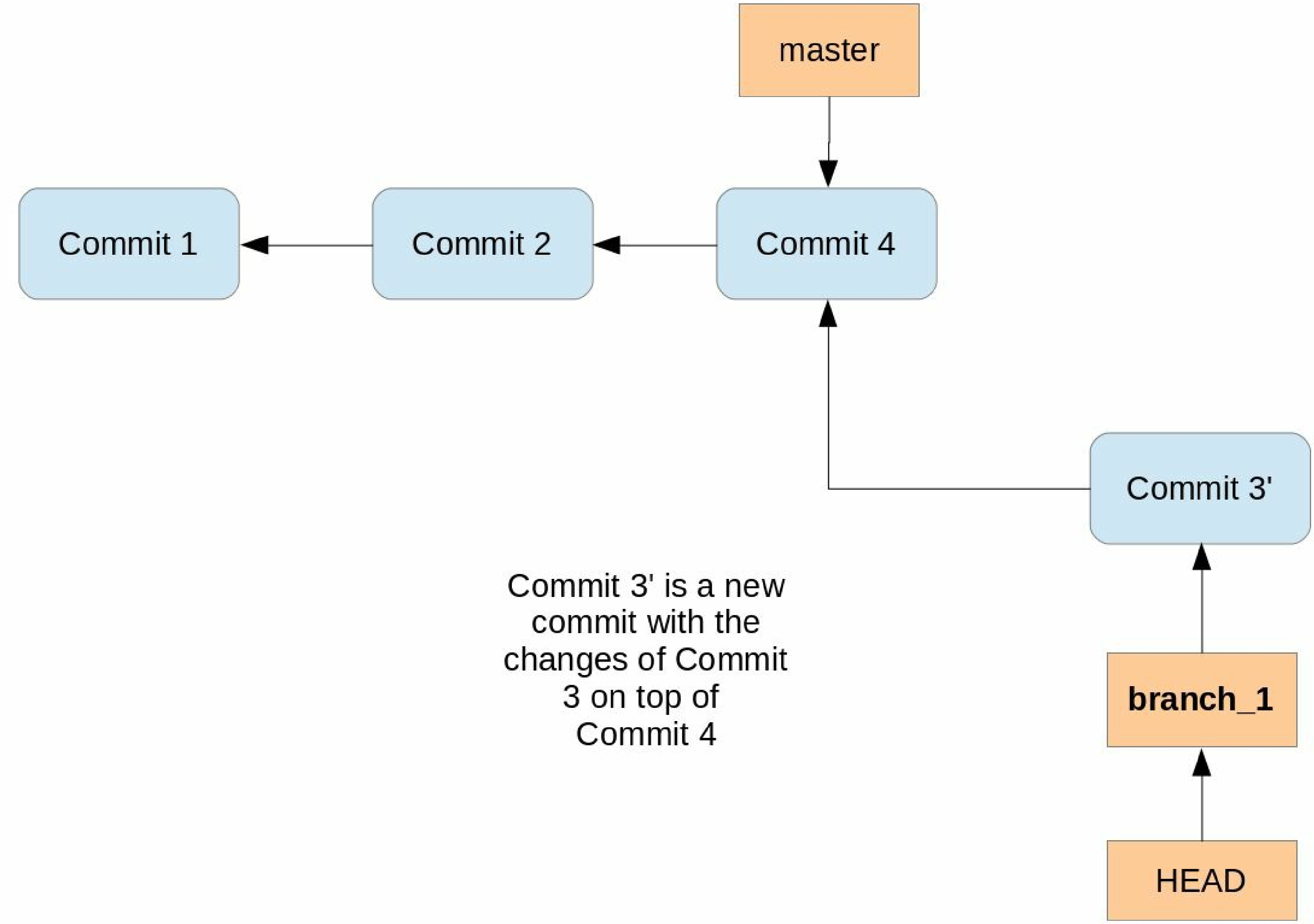
46.1. Rebasing branches

You can use Git to rebase one branch on another one. As described, the `merge` command combines the changes of two branches. If you rebase a branch called A onto another, the `git` command takes the changes introduced by the commits of branch A and applies them based on the HEAD of the other branch. This way the changes in the other branch are also available in branch A.

The process is displayed in the following picture. We want to rebase the branch called `branch_1` onto master.



Running the rebase command creates a new commit with the changes of the branch on top of the master branch.



Performing a rebase does not create a merge commit. The final result for the source code is the same as with merge but the commit history is cleaner; the history appears to be linear.

Rebase can be used to forward-port a feature branch in the local Git repository onto the changes of the master branch. This ensures that your feature is close to the tip of the upstream branch until it is finally published.

If you rewrite more than one commit by rebasing, you may have to solve conflicts per commit. In this case the merge operations might be simpler to be performed because you only have to solve merge conflicts once.

Also, if your policy requires that all commits result in correct software you have to test all the rewritten commits since they are "rewritten" by the rebase algorithm. Since merge/rebase/cherry-pick are purely text-based and do not understand the semantics of these texts they can end up with logically incorrect results. Hence, it might be more efficient to merge a long feature branch into upstream instead of rebasing it since you only have to review and test the merge commit.

Note

You can use the rebase command to change your Git repository history commits. This is called *interactive* rebase, see Section 51.1, “Editing history with the interactive rebase” for information about this feature.

46.2. Good practice for rebase

You should avoid using the Git rebase operation for changes which have been published in other Git repositories. The Git rebase operation creates new commit objects, this may confuse other developers using the existing commit objects.

Assume that a user has a local feature branch and wants to push it to a branch on the remote repository. However, the branch has evolved and therefore pushing is not possible. Now it is good practice to fetch the latest state of the branch from the remote repository. Afterwards you rebase the local feature branch onto the remote tracking branch. This avoids an unnecessary merge commit. This rebasing of a local feature branch is also useful to incorporate the latest changes from remote into the local development, even if the user does not want to push right away.

Tip

Rebasing and amending commits is safe as long as you do not push any of the changes involved in the rebase. For example, when you cloned a repository and worked in this local repository. Rebasing is a great way to keep the history clean before contributing back your modifications.

Warning

In case you want to rewrite history for changes you have shared with others you need to use the *-f* parameter in your `git push` command and subsequently your colleagues have to use fetch -f to fetch the rewritten commits.

```
# using forced push
git push -f
```

46.3. Example for a rebase

The following demonstrates how to perform a rebase operation.

```
# create new branch
git checkout -b rebasetest

# create a new file and put it under revision control
touch rebase1.txt
git add . && git commit -m "work in branch"

# do changes in master
git checkout master

# make some changes and commit into testing
echo "rebase this to rebasetest later" > rebasefile.txt
git add rebasefile.txt
git commit -m "create new file"

# rebase the rebasetest onto master
git checkout rebasetest
git rebase master

# now you can fast forward your branch onto master
git checkout master
git merge rebasetest
```

Chapter 47. Selecting individual commits with cherry-pick

This chapter explains how to use the git cherry-pick command.

47.1. Applying a single commit

The `git cherry-pick` command allows you to select the patch which was introduced with an individual commit and apply this patch on another branch. The patch is captured as a new commit on the other branch.

This way you can select individual changes from one branch and transfer them to another branch.

> **Note**
>
> The new commit does not point back to its original commit so do not use cherry-pick blindly since you may end up with several copies of the same change. Most often cherry-pick is either used locally (to emulate an interactive rebase) or to port individual bug fixes done on a development branch into maintenance branches.

47.2. Example: Using cherry-pick

In the following example you create a new branch and commit two changes.

```
# create new branch
git checkout -b picktest

# create some data and commit
touch pickfile.txt
git add pickfile.txt
git commit -m "adds new file"

# create second commit
echo "changes to file" > pickfile.txt
git commit -a -m "changes in file"
```

You can check the commit history, for example, with the `git log --oneline` command.

```
# see change commit history

git log --oneline

# results in the following output

2fc2e55 changes in file
ebb46b7 adds new file
[MORE COMMITS]
330b6a3 initial commit
```

The following command selects the first commit based on the commit ID and applies its changes to the master branch. This creates a new commit on the master branch.

```
git checkout master
git cherry-pick ebb46b7
```

The `cherry-pick` command can be used to change the order of commits. `git cherry-pick` also accepts commit ranges for example in the following command.

```
git checkout master
# pick the last two commits
git cherry-pick picktest~1..picktest~2
```

> **Tip**
>
> See Section 7.5, "Commit ranges with the double dot operator" for more information about commit ranges.

If things go wrong or you change your mind, you can always reset to the previous state using the following command.

```
git cherry-pick --abort
```

Part XVIII. Solving conflicts during a merge or a rebase operation

Chapter 48. Handling merge conflicts

This chapter describes how to resolve conflicts during merge operations.

48.1. What is a conflict during a merge operation?

A conflict during a merge operation occurs if two commits from different branches have modified the same content and Git cannot automatically determine how both changes should be combined when merging these branches.

This happens for example if the same line in a file has been replaced by two different commits.

If a conflict occurs, Git marks the conflict in the file and the programmer has to resolve the conflict manually.

After resolving it, he adds the file to the staging area and commits the change. These steps are required to finish the merge operation.

48.2. Keep a version of a file during a merge conflict

Sometimes if a conflict occurs the developer does not want to solve the conflict. He decides that he wants to keep the original version or the new version of the file.

For this, there is the `--theirs` and the `--ours` options on the `git checkout` command. The first option keeps the version of the file that you merged in, and the second option keeps the version before the merge operation was started.

```
git checkout --ours foo/bar.java
git add foo/bar.java
```

```
git checkout --theirs foo/bar.java
git add foo/bar.java
```

Chapter 49. Exercise: Solving a conflict during a merge operation

49.1. Create a conflict

In the following example you create a conflict during a merge operation.

The following steps create a merge conflict. It assumes that *repo1* and *repo2* have the same *origin* repository defined.

```
# switch to the first directory
cd ~/repo01
# make changes
echo "Change in the first repository" > mergeconflict.txt
# stage and commit
git add . && git commit -a -m "Will create conflict 1"

# switch to the second directory
cd ~/repo02
# make changes
touch mergeconflict.txt
echo "Change in the second repository" > mergeconflict.txt
# stage and commit
git add . && git commit -a -m "Will create conflict 2"
# push to the master repository
git push

# switch to the first directory
cd ~/repo01

# now try to push from the first directory
# try to push --> assuming that the same remote repository is used,
# you get an error message
git push
```

As this push would not result in a non-fast-format merge, you receive an error message similar to the following listing.

```
! [rejected]        master -> master (fetch first)
error: failed to push some refs to '../remote-repository.git/'
hint: Updates were rejected because the remote contains work that you do
hint: not have locally. This is usually caused by another repository pushing
hint: to the same ref. You may want to first integrate the remote changes
hint: (e.g., 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

To solve this, you need to integrate the remote changes into your local repository. In the following listing the `git fetch` command gets the changes from the remote repository. The `git merge` command tries to integrate it into your local repository.

```
# get the changes via a fetch
git fetch origin

# now merge origin/master into the local master
# this creates a merge conflict in your
# local repository
git merge origin/master
```

This creates the conflict and a message similar to the following.

```
Auto-merging mergeconflict.txt
CONFLICT (add/add): Merge conflict in mergeconflict.txt
Automatic merge failed; fix conflicts and then commit the result.
```

The resulting conflict is displayed in Section 49.2, “Review the conflict in the file” and solved in Section 49.3, “Solve a conflict in a file”

Tip

If you use the `git pull` command it performs the "fetch and merge" or the "fetch and rebase" command together in one step. Whether merge or rebase is used depends on your Git configuration for the branch. See Section 10.4, “Avoid merge commits for pulling” for the global configuration.

49.2. Review the conflict in the file

Git marks the conflicts in the affected files. In the example from Section 49.1, “Create a conflict” one file has a conflict and the file looks like the following listing.

```
<<<<<<< HEAD
Change in the first repository
=======
Change in the second repository
>>>>>>> b29196692f5ebfd10d8a9ca1911c8b08127c85f8
```

The text above the ======= signs is the conflicting change from your current branch and the text below is the conflicting change from the branch that you are merging in.

49.3. Solve a conflict in a file

In this example you resolve the conflict which was created in Section 49.1, “Create a conflict” and apply the change to the Git repository.

To solve the merge conflict you edit the file manually. The following listing shows a possible result.

```
Change in the first and second repository
```

Afterwards add the affected file to the staging area and commit the result. This creates the merge commit. You can also push the integrated changes now to the remote repository.

```
# add the modified file
git add .

# creates the merge commit
git commit -m "Merge changes"

# push the changes to the remote repository
git push
```

Instead of using the *-m* option in the above example you can also use the `git commit` command without this option. In this case the command opens your default editor with the default commit message about the merged conflicts. It is good practice to use this message.

> **Tip**
>
> Alternatively, you could use the `git mergetool` command. `git mergetool` starts a configurable merge tool that displays the changes in a split screen. Some operating systems may come with a suitable merge tool already installed or configured for Git.

Chapter 50. Handling rebase conflicts

This chapter describes how to resolve conflicts during rebase operations.

50.1. What is a conflict during a rebase operation?

During a rebase operaton, several commits are applied onto a certain commit. If you rebase a branch onto another branch, this commit is the last common ancestor of the two branches.

For each commit which is applied it is possible that a conflict occurs.

50.2. Handling a conflict during a rebase operation

If a conflict occurs during a rebase operation, the rebase operation stops and the developer needs to resolve the conflict. After he has solved the conflicts, the developer instructs Git to continue with the rebase operation.

A conflict during a rebase operation is solved similarly to the way a conflict during a merge operation is solved. The developer edits the conflicts and adds the files to the Git index. Afterwards he continues the rebase operation with the following command.

```
# rebase conflict is fixed, continue with the rebase operation
git rebase --continue
```

To see the files which have a rebase conflict use the following command.

```
# lists the files which have a conflict
git diff --name-only --diff-filter=U
```

You solve such a conflict similar to the description in Section 49.3, “Solve a conflict in a file”.

You can also skip the commit which creates the conflict.

```
# skip commit which creates the conflict
git rebase --skip
```

50.3. Aborting a rebase operation

You can also abort a rebase operation with the following command.

```
# abort rebase and recreate the situation before the rebase
git rebase --abort
```

50.4. Picking theirs or ours for conflicting file

If a file is in conflict you can instruct Git to take the version from the new commit of the version of commit onto which the new changes are applied. This is sometimes easier than to solve all conflicts manually. For this you can use the `git checkout` with the *`--theirs`* or *`--ours`* flag. During the conflict *`--ours`* points to the file in the commit onto which the new commit is placed, i.g., using this skips the new changes for this file.

Therefore to ignore the changes in a commit for a file use the following command.

```
git checkout --ours foo/bar.java
git add foo/bar.java
```

To take the version of the new commit use the following command.

```
git checkout --theirs foo/bar.java
git add foo/bar.java
```

Part XIX. Rewriting history in the repository

Chapter 51. Interactive rebase

51.1. Editing history with the interactive rebase

Git allows you to edit your commit history with a functionality called `interactive rebase`. For example, you can combine several commits into one commit, reorder or skip commits and edit the commit message.

This is useful as it allows the user to rewrite some commit history (cleaning it up) before pushing the changes to a remote repository.

Interactive rebase allows you to quickly edit a series of commits using the following actions:

Table 51.1. Interactive rebase actions

Action	Description
pick	includes the selected commit, moving pick entries enables reordering of commits
skip	removes a commit
reword	similar to pick but allows modifying the commit message
edit	amends the commit
squash	combines the changes of the commit with the previous commit and combines their commit messages
fixup	squashes the changes of a commit into the previous commit discarding the squashed commit's message

The setup for the rebase is called the *rebase plan*. Based on this plan, the actual interactive rebase can be executed.

Warning

It is safe to use interactive rebase as long as the commits have not been pushed to another repository. As the interactive rebase creates new commit objects, other developers might be confused if you rebase already published changes.

51.2. Example: Interactive rebase

The following commands create several commits which will be used for the interactive rebase.

```
# create a new file
touch rebase.txt

# add it to git
git add . && git commit -m "add rebase.txt  to staging area"

# do some silly changes and commit
echo "content" >> rebase.txt
git add . && git commit -m "add content"
echo " more content" >> rebase.txt
git add . && git commit -m "just testing"
echo " more content" >> rebase.txt
git add . && git commit -m "woops"
echo " more content" >> rebase.txt
git add . && git commit -m "yes"
echo " more content" >> rebase.txt
git add . && git commit -m "add more content"
echo " more content" >> rebase.txt
git add . && git commit -m "creation of important configuration file"

# check the git log message
git log
```

We want to combine the last seven commits. You can do this interactively via the following command.

```
git rebase -i HEAD~7
```

This command opens your editor of choice and lets you configure the rebase operation by defining which commits to *pick*, *squash* or *fixup*.

The following listing shows an example of the selection, we pick the last commit, squash 5 commits and fix the sixth commit. The listing uses the long format of the commands (for example `fixup` instead of the short form `f`) for better readability.

```
pick 7c6472e rebase.txt added to index
fixup 4f73e68 added content
fixup bc9ec3f just testing
fixup 701cbb5 ups
fixup 910f38b yes
fixup 31d447d added more content
squash e08d5c3 creation of important configuration file

# Rebase 06e7464..e08d5c3 onto 06e7464
#
# Commands:
#  p, pick = use commit
#  r, reword = use commit, but edit the commit message
#  e, edit = use commit, but stop for amending
#  s, squash = use commit, but meld into previous commit
#  f, fixup = like "squash", but discard this commit's log message
#  x, exec = run command (the rest of the line) using shell
#
# These lines can be re-ordered; they are executed from top to bottom.
#
# If you remove a line here THAT COMMIT WILL BE LOST.
# However, if you remove everything, the rebase will be aborted.
```

Chapter 52. Rewriting commit history with git filter-branch

52.1. Using git filter-branch

The `git filter-branch` command allows you to rewrite the Git commit history for selected branches and to apply custom filters on each revision. This creates different hashes for all modified commits. This implies that you get new IDs for all commits based on any rewritten commit.

The command allows you to filter for several values, e.g., the author, the message, etc. For details please see the following link:

git-filter-branch(1) Manual Page

> **Warning**
>
> Using the `filter-branch` command is dangerous as it changes the Git repository. It changes the commit IDs and reacting on such a change requires explicit action from the developer, e.g., trying to rebase the stale local branch onto the corresponding rewritten remote-tracking branch.

A practical case for using `git filter-branch` is where you have added a file which contains a password or a huge binary file to the Git repository, and you want to remove this file from the history. To completely remove the file you need to run the `filter-branch` command on all branches.

52.2. filter-branch example

The following listing shows an example on how to replace the email address from one author of all the commits via the `git filter-branch` command.

```
git filter-branch -f \
--env-filter 'if [ "$GIT_AUTHOR_NAME" = "Lars Vogel" ]; then \
GIT_AUTHOR_EMAIL="lars.vogel@gmail.com"; fi' HEAD)
```

Part XX. Including other repositories via Git submodules

Chapter 53. Git submodules

53.1. What are submodules?

Git allows you to include other Git repositories into a Gitrepository. This is useful in case you want to include a certain library in another repository or in case you want to aggregate certain Git repositories.

Git calls these included Git repositories *submodules*. Git allows you to commit, pull and push to these repositories independently.

53.2. Adding a submodule to a Git repository

You add a submodule to a Git repository via the `git submodule add` command. The `git submodule init` command creates the local configuration file for the submodules if this configuration does not exist.

```
# add a submodule to your Git repo
git submodule add [URL to Git repo]

# initialize submodule configuration
git submodule init
```

Chapter 54. Working with submodules

54.1. Updating submodules

To pull in changes into a Git repository including the changes in submodules, you can use the `--` *`recurse-submodules`* parameter in the `git pull command`.

```
# pull in the changes from main repo and submodules
git pull --recurse-submodules
```

Use the `git submodule update` command to set the submodules to the commit specified by the main repository.

```
# setting the submodules to the commit defined by master
git submodule update
```

> **Warning**
>
> The fact that submodules track commits and not branches frequently leads to confusion. That is why Git 1.8.2 added the option to also track branches. Read the following sections to learn more about this.

54.2. Tracking branches with submodules

Since its 1.8.2 release the Git system allows tracking a branch in a submodule. To track branches you specify the branch with the *-b* parameter during the `submodule add` command.

This allows you use to use *--remote* parameter in the `git submodule update` command.

```
# add submodule to track master branch
git submodule add -b master [URL to Git repo]

# update your submodule
# --remote will also fetch and ensure that
# the latest commit from the branch is used
git submodule update --remote

# to avoid fetching use
git submodule update --remote --no-fetch
```

54.3. Tracking commits

Without any additional parameter, submodules are tracked by commit, i.e., the main Git repository remembers a certain commit of the submodule.

The `git submodule update` command sets the Git repository of the submodule to that particular commit. The submodule repository tracks its own content which is nested into the main repository. This main repository refers to a commit of the nested submodule repository.

> **Warning**
>
> This means that if you pull in new changes into the submodules, you need to create a new commit in your main repository in order to track the updates of the nested submodules.

If you update your submodule and want to use this update in your main repository, you need to commit this change in your main repository. The `git submodule update` command sets the submodule to the commit referred to in the main repository.

The following example shows how to update a submodule to its latest commit in its master branch.

```
# update submodule in the master branch
# skip this if you use --recurse-submodules
# and have the master branch checked out
cd [submodule directory]
git checkout master
git pull

# commit the change in main repo
# to use the latest commit in master of the submodule
cd ..
git add [submodule directory]
git commit -m "move submodule to latest commit in master"

# share your changes
git push
```

Another developer can get the update by pulling in the changes and running the submodules update command.

```
# another developer wants to get the changes
git pull

# this updates the submodule to the latest
# commit in master as set in the last example
git submodule update
```

> **Warning**
>
> With this setup you are tracking commits, so if the master branch in the submodule moves on, you are still pointing to the existing commit. You need to repeat this procedure every time you want to use new changes of the submodules. See Section 54.2, "Tracking branches with submodules" for tracking branches.

Part XXI. Git alias, bisect and patch

Chapter 55. Git alias

55.1. Using an alias

An *alias* in Git allows you to create a short form of one or several existing Git commands. For example, you can define an alias which is a short form of your own favorite commands or you can combine several commands with an alias.

55.2. Alias examples

The following defines an *alias* to see the staged changes with the new `git staged` command.

```
git config --global alias.staged 'diff --cached'
```

Or you can define an *alias* for a detailed `git log` command. The following command defines the `git ll` *alias*.

```
git config --global alias.ll 'log --graph --oneline --decorate --all'
```

You can also run external commands. In this case you start the *alias* definition with a `!` character. For example, the following defines the `git ac` command which combines `git add . -A` and `git commit` commands.

```
# define alias
git config --global alias.act '!git add . -A && git commit'

# to use it
git act -m "message"
```

> **Warning**
>
> In the past *msysGit* for Windows had problems with an *alias* beginning with `!`, but it has been reported that this now works with *msysGit*, too .

Chapter 56. Error search with git bisect

56.1. Using git bisect

The `git bisect` command allows you to run a binary search through the commit history to identify the commit which introduced an issue. You specify a range of commits and a script that the `bisect` command uses to identify whether a commit is good or bad.

This script must return 0 if the condition is fulfilled and non-zero if the condition is not fulfilled.

56.2. git bisect example

Create a new Git repository, create the *text1.txt* file and commit it to the repository. Do a few more changes, remove the file and again do a few more changes.

We use a simple shell script which checks the existence of a file. Ensure that this file is executable.

```
#!/bin/bash
FILE=$1

if [ -f $FILE ];
then
   exit 0;
else
   exit 1;
fi
```

Afterwards use the `git bisect` command to find the bad commit. First you use the *git bisect start* command to define a commit known to be bad (showing the problem) and a commit known to be good (not showing the problem).

```
# define that bisect should check
# the last 5 commits
git bisect start HEAD HEAD~5
```

Afterwards run the bisect command using the shell script.

```
# assumes that the check script
# is a directory above the current
git bisect run ../check.sh test1.txt
```

> **Tip**
>
> The above commands serve as an example. The existence of a file can be easier verified with the `git bisect` command: `git bisect run test -f test1.txt`

Chapter 57. Git patch

57.1. What is a patch file?

A *patch* is a text file that contains changes to the source code. A patch created with the `git format-patch` command includes meta-information about the commit (committer, date, commit message, etc) and also contains the changes introduced in binary data in the commit, for example, an image.

This file can be sent to someone else and this developer can use this file to apply the changes to his local repository. The metadata is preserved.

Alternatively you could create a diff file with the `git diff` command, but this diff file does not contain the metadata information.

57.2. Create and apply patches

The following example creates a branch, changes several files and creates a commit recording these changes.

```
# create a new branch
git branch mybranch
# use this new branch
git checkout mybranch
# make some changes
touch test05
# change some content in an existing file
echo "new content for test01" >test01
# commit this to the branch
git add .
git commit -m "first commit in the branch"
```

The next example creates a patch for these changes.

```
# creates a patch --> git format-patch master
git format-patch origin/master

# this creates the file:
# patch 0001-First-commit-in-the-branch.patch
```

To apply this patch to your master branch in a different clone of the repository, switch to it and use the `git apply` command.

```
# switch to the master branch
git checkout master

# apply the patch
git apply 0001-First-commit-in-the-branch.patch
```

Afterwards you can commit the changes introduced by the patches and delete the patch file.

```
# patch is applied to master
# change can be committed
git add .
git commit -m "apply patch"

# delete the patch file
rm 0001-First-commit-in-the-branch.patch
```

> **Tip**
>
> Use the `git am` command to apply and commit the changes in a single step. To apply and commit all patch files in the directory use, for example, the `git am *.patch` command. You specify the order in which the patches are applied by specifying them on the command line.

57.3. Create a patch for a selected commit

You can specify the commit ID and the number of patches which should be created. For example, to create a patch for selected commits based on the HEADpointer you can use the following commands.

```
# create patch for the last commit based on HEAD
git format-patch -1 HEAD

# create a patch series for the last three commits
# based on head
git format-patch -3 HEAD
```

Part XXII. Using GitHub and Bitbucket for hosting

Chapter 58. Authentication via SSH

58.1. The concept of SSH

Most Git (and Gerrit) servers support SSH based authentication. Thisrequires a *SSH key pair* for automatic authentication.

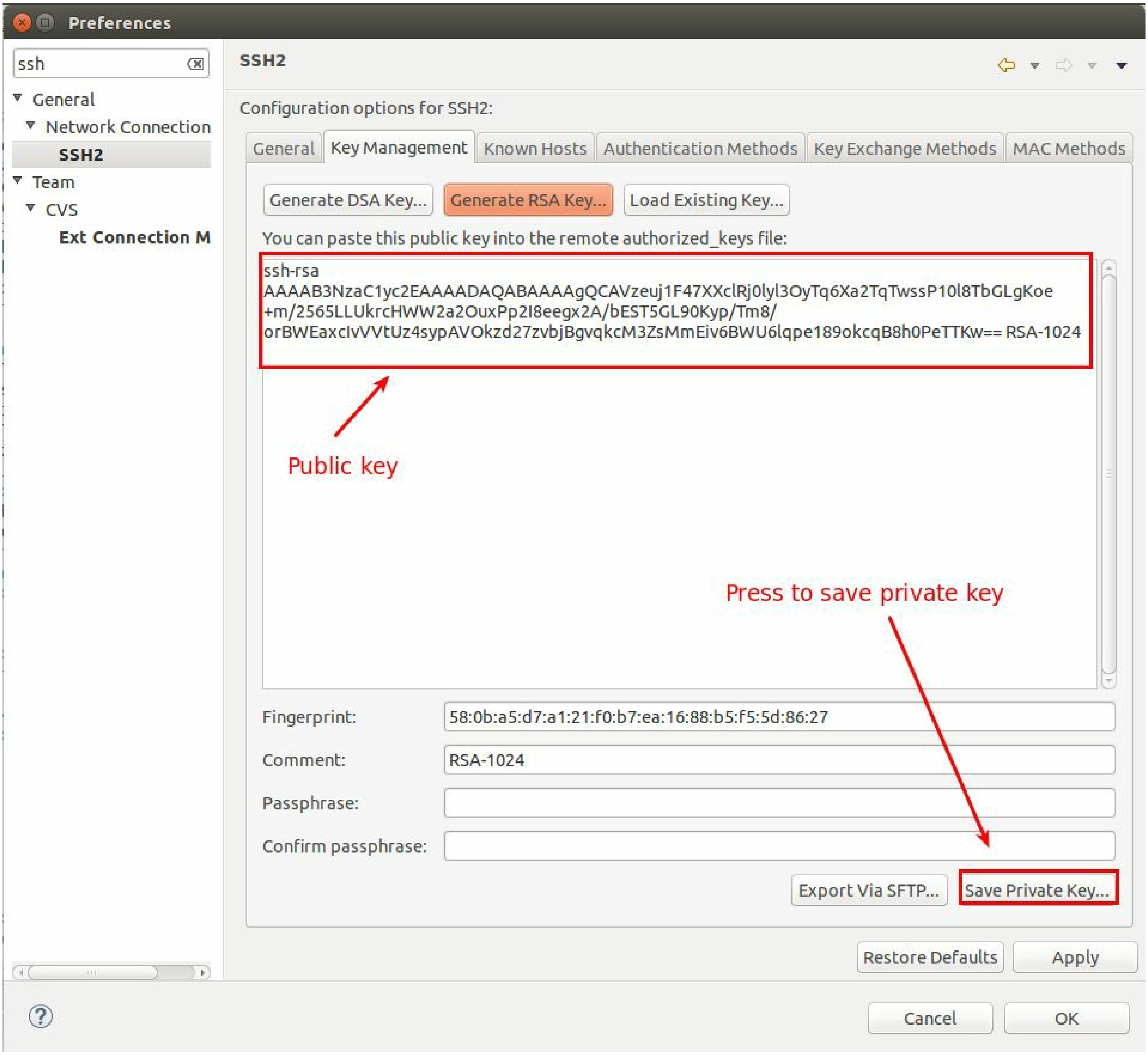
An SSH key par consists of a public and private key. The publickey is uploaded to the application you want to authenticate with. The application has no access to the private key. If you interact with the hosting provider via the ssh protocol, the public key is used to identify a user who encrypted the data during communication with the corresponding private key.

58.2. SSH key pair generation

To create an SSH key under Linux (or Windows / Mac with OpenSSHinstalled) switch to the command line and execute the following commands. The generated SSHkey is by default located in the *.ssh* directory of the user home directory. Ensure that you backup existing keys in this directory before running the following commands.

```
# Switch to your .ssh directory
cd ~/.ssh

# If the directory
# does not exist, create it via:
# mkdir .ssh

# Manually backup all existing content of this dir!!!

# Afterwards generate the ssh key
ssh-keygen -t rsa -C "your_email@youremail.com"

# Press enter to select the default directory
# You will be prompted for an optional passphrase
# A passphrase protects your private key
# but you  have to enter it manually during ssh operations
```

The Eclipse IDE allows you to create an SSH key pair viaWindow → Preferences → General → Network Connection → SSH2.



It is good practice to use a passphrase to protect your private key. It is also good practice to use operating system level permission settings to ensure that only the owning user can access the `~/.ssh` folder and its content.

Note

In the above `ssh-keygen` command the -C parameter is a comment. Using your email is good practice so that someone looking at your public key can contact you in case they have questions. Including the email enables system administrators to contact the person in case of questions.

The result will be two files, *`id_rsa`* which is your private key and *`id_rsa.pub`* which is your public key.

You find more details for the generation of an SSH key on the following webpages: [GitHub Help: description of SSH key creation]() or [OpenSSH manual]().

Tip

You can specify alternative key names with the `-f` parameter on the command line. This is helpful if you have multiple different repositories and you want to have a different key for each one. For example, you can name your SSH keys in domainname format, e.g., *eclipse.org* and *eclipse.org.pub* as well as *github.com* and *github.com.pub*.

You need additional configuration in the *.ssh/config* file, because only the *id_rsa* will be picked up by default. The following code shows an example.

```
Host *.eclipse.org
  IdentityFile ~/.ssh/eclipse.org

Host *.github.com
  IdentityFile ~/.ssh/github.com
```

Chapter 59. GitHub

59.1. What is GitHub?

GitHub is a popular hosting provider for Git repositories. GitHub provides also additional services around these repositories, for example an issue tracker for each repository, build server integration and more.

GitHub supports that repositories can be cloned to a new Git repository hosted at Github. GitHub uses the term *fork* or *forking* for creating such clones.

GitHub provides free hosting for publicly visible Git repositories. A public repository can be cloned by other people at any point in time.

If the repository should not be visible to everyone, Git allows to create private repositories, but you must pay for this service a monthly rate. Private repository allows you to specify the people which have access to the repository and to define their access rights.

GitHub can be found under the following URL.

GitHub

If you create an account at GitHub, you can create a repository. After creating a repository at GitHub, you will get a description of all the commands you need to execute to upload your project to GitHub. Follow the instructions below.

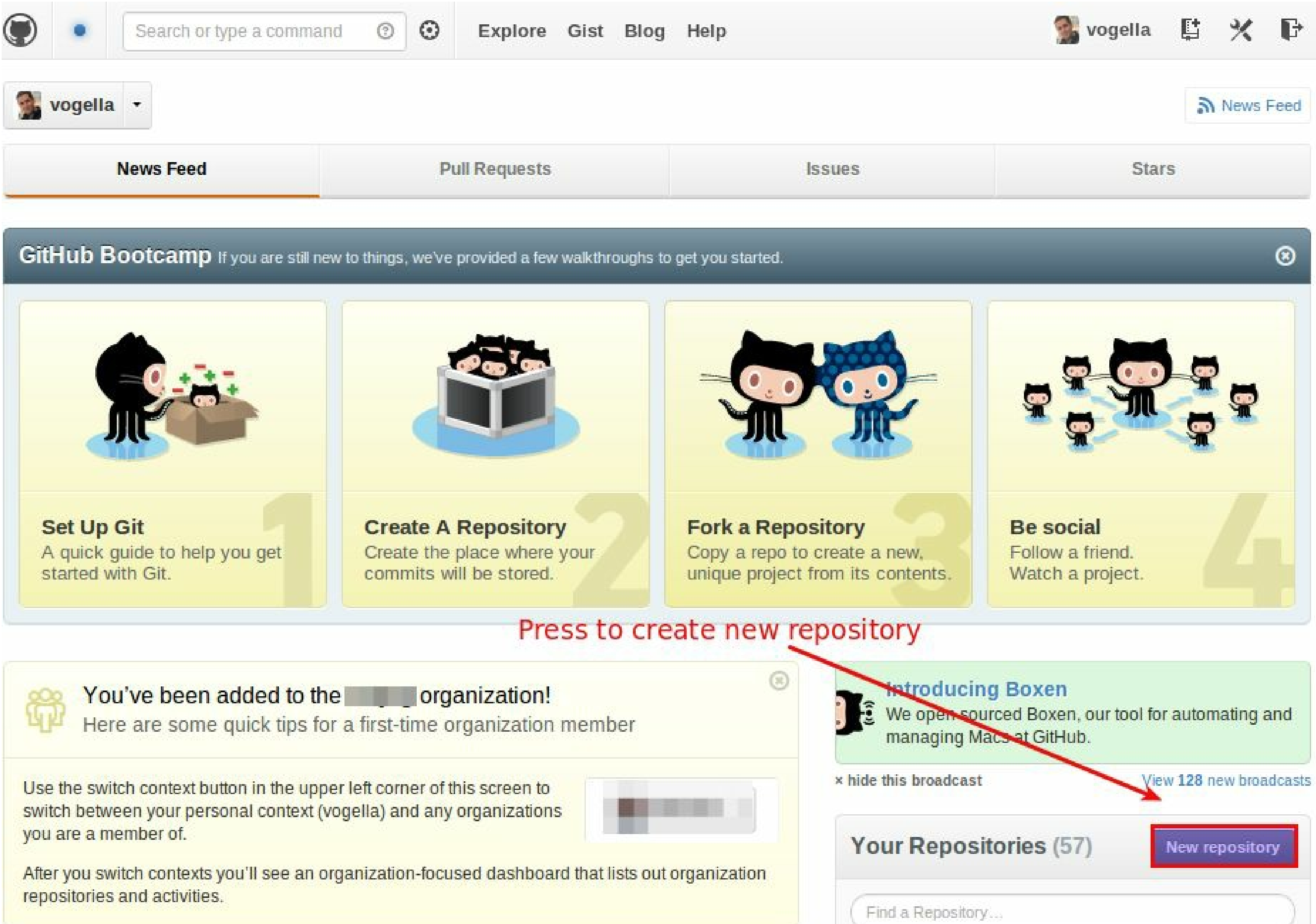
These instructions will be similar to the following commands.

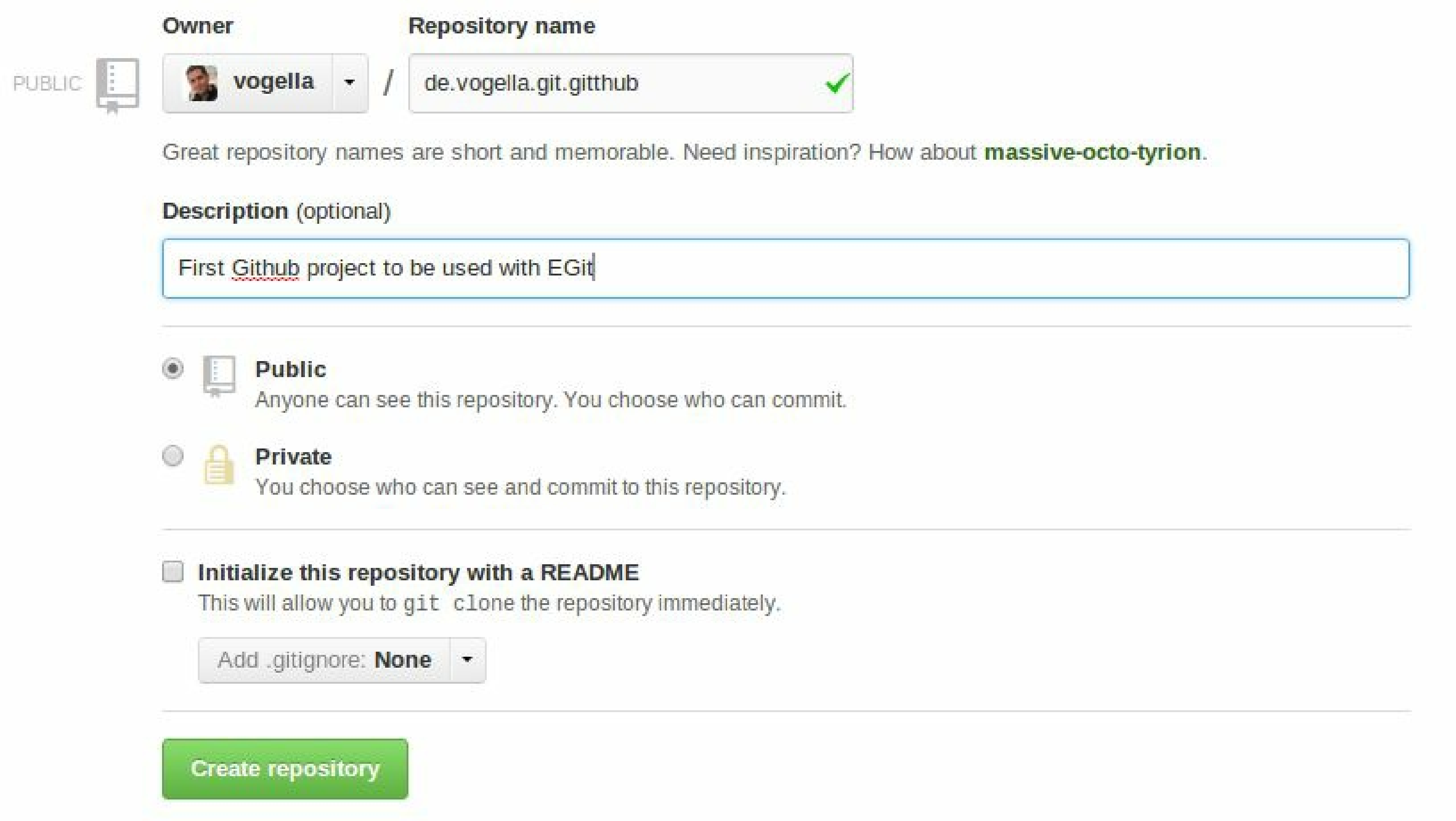
```
# global setup:
#  set up git
  git config --global user.name "Your Name"
  git config --global user.email your.email@gmail.com

# next steps for a new repository
  mkdir gitbook
  cd gitbook
  git init
  touch README
  git add README
  git commit -m 'first commit'
  git remote add origin git@github.com:vogella/gitbook.git
  git push -u origin master

# alternatively for an existing Git repo
# add remote and push
  cd existing_git_repo
  git remote add origin git@github.com:vogella/gitbook.git
  git push -u origin master
```

GitHub allows you to use SSH based or HTTPS based authentication to access your repositories. To clone, pull or fetch from a public available repository no authentication is required.

59.2. Create repository in GitHub

Once you create a user at Github, you can create a new public repository. For example the following screenshots demonstrate the creation of the *de.vogella.git.github* repository.





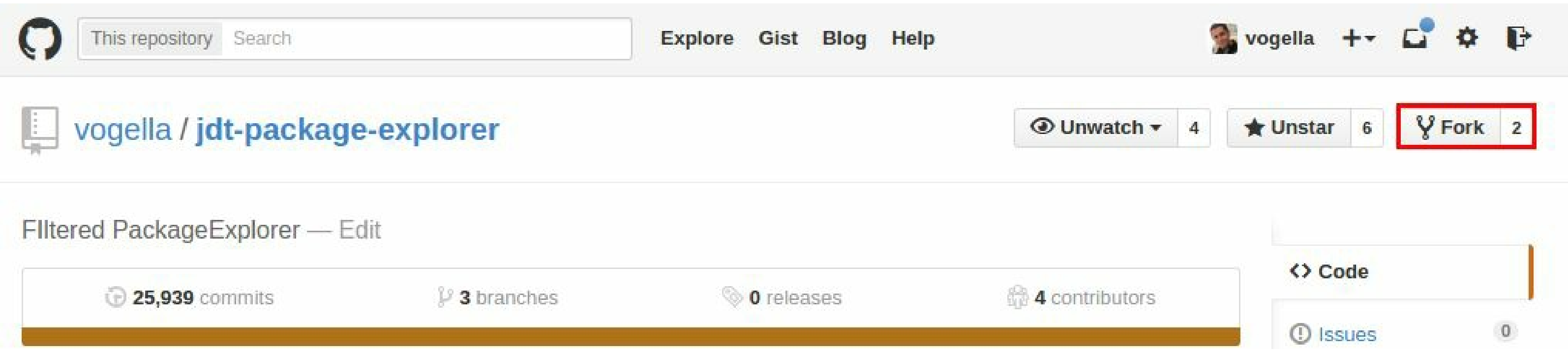
After creation of your new repository GitHub displays the information what you have to do if you want to connect to this repository via the command line. As we are going to use EGit you can ignore this information.

59.3. Merging pull request at GitHub

GitHub uses *Pull requests* for contributions.

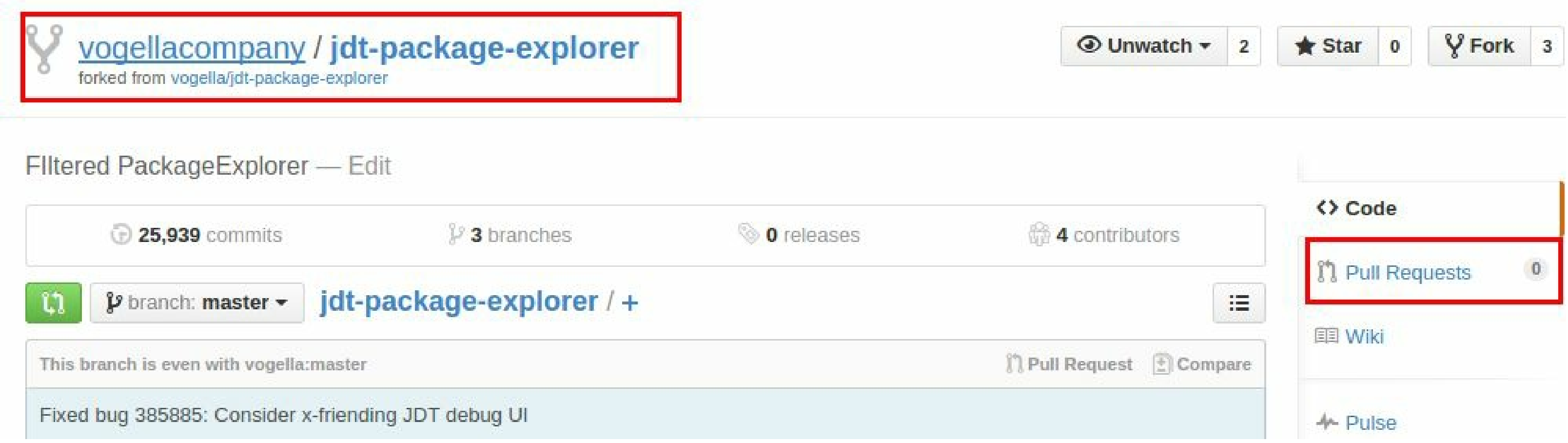
The typical workflow in GitHub is to fork a repository, create changes in your fort and send a pull request to the origin repository via the GitHub webinterface.

GitHub makes is easy to fork a repository via its web interface. Simply click the *Fork* button of a repository of your choice.

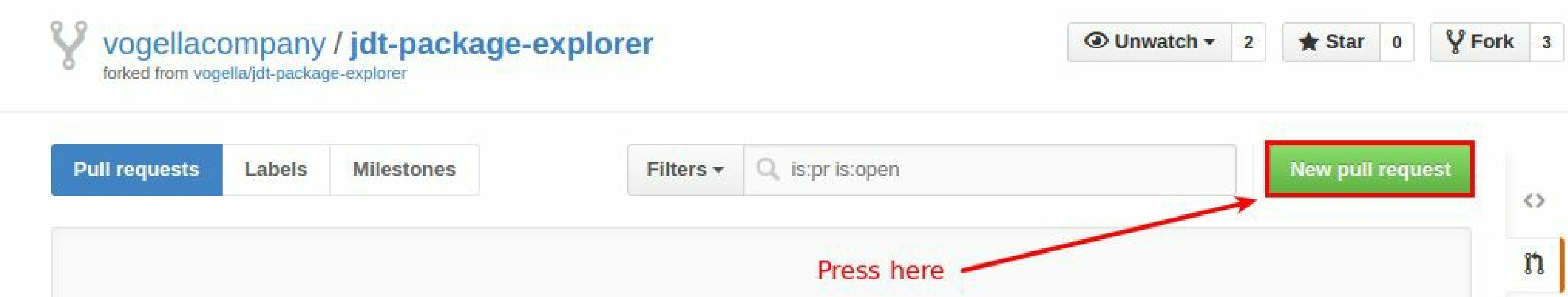


You can now clone this fork to your local development environment and push the changes to this fork at GitHub.

Afterwards you can create a pull request for the repository you forked from. Your repository is the fork as highlighted in the screenshot. Press *Pull Requests* to see existing and create new pull requests.

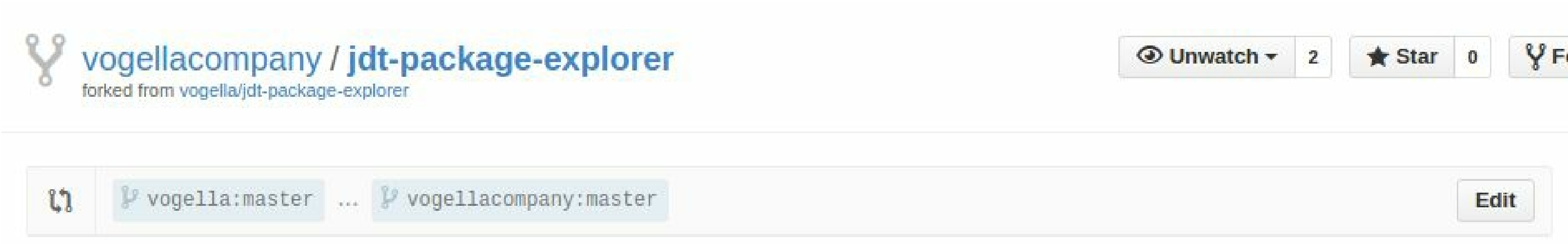


Click *New pull request* to create a new one.



On the next screen you can specify the direction of the pull request and the branches if you select the

Edit button.



If the owner of the repository accepts your pull request your changes are integrated into the original repository.

Chapter 60. Bitbucket

60.1. What is Bitbucket?

Bitbucket offers free hosting of public and private Git repositories.

Bitbucket allows unlimited public and private repositories. The number of participants for a free private repository is currently limited to 5 collaborators, i.e., if you have more than 5 developers which need access to a private repository you have to pay money to BitBucket.

60.2. Creating a repository

You need to create a user via the web interface of Bitbucket. After creating this user you can create new repositories via the web interface.

After creating a new repository on BitBucket, you can use the following instructions connect a local Git repository with the BitBucket repository.

These instructions will be similar to the following commands.

```
# Global setup:
# Set up git
  git config --global user.name "Your Name"
  git config --global user.email your.email@gmail.com

# Next steps for a new repository
  mkdir gitbook
  cd gitbook
  git init
  touch README
  git add README
  git commit -m 'first commit'
  git remote add origin ssh://git@bitbucket.org/vogella/gitbook.git
  git push -u origin master

# alternatively for an existing Git repo
# add remote and push
  cd existing_git_repo
  git remote add origin ssh://git@bitbucket.org/vogella/gitbook.git
  git push -u origin master
```

Part XXIII. Running your own Git server

Chapter 61. Own Git server

61.1. Hosting your own Git server

As described before, you do not need a server. You can just use a file system or a public Git provider, such as GitHub or Bitbucket. Sometimes, however, it is convenient to have your own server, and installing it under Ubuntu is relatively easy.

First make sure you have installed the SSH tooling.

```
sudo apt-get install ssh
```

If you have not yet installed Git on your server, you need to do this too.

```
sudo apt-get install git-core
```

Create a new user and set a password for the Git system.

```
sudo adduser git
```

Now log in with your Git user and create a bare repository.

```
# Login to server
# to test use localhost
ssh git@IP_ADDRESS_OF_SERVER

# Create repository
git init --bare example.git
```

Now you can push to the remote repository.

```
mkdir gitexample
cd gitexample
git init
touch README
git add README
git commit -m 'first commit'
git remote add origin git@IP_ADDRESS_OF_SERVER:example.git
git push origin master
```

61.2. Give write access to a Git repository

The typical setup based on the created "git" user from above is that the public SSH key of each user is added to the *~/.ssh/authorized_keys* file of the "git" user. Afterwards everyone can access the system using the "git" user.

Alternatively you could use LDAP authentication or other special configurations.

61.3. Security setup for the git user

The Git installation provides a specialized shell, which can be assigned to the user. Typically this shell is located under in */usr/bin/git-shell* and can be assigned to the user via the */etc/passwd* configuration file to the Git user. If you assign this shell to the Git user, this user can also perform git commands which add safety to your Git setup.

Chapter 62. Git commit hooks

62.1. Usage of Git hooks

Git provides commit hooks, e.g., programs which can be executed at a pre-defined point during the work with the repository. For example, you can ensure that the commit message has a certain format or trigger an action after a push to the server.

These programs are usually scripts and can be written in any language, e.g., as shell scripts or in Perl, Python etc. You can also implement a hook, for example, in C and use the resulting executables. Git calls the scripts based on a naming convention.

62.2. Client and server side commit hooks

Git provides hooks for the client and for the server side. On the server side you can use the *pre-receive* and *post-receive* script to check the input or to trigger actions after the commit. The usage of a server commit hook requires that you have access to the server. Hosting providers like GitHub or Bitbucket do not offer this access.

If you create a new Git repository, Git creates example scripts in the *.git/hooks* directory. The example scripts end with *.sample*. To activate them make them executable and remove the *.sample* from the filename.

The hooks are documented under the following URL: Git hooks manual page.

62.3. Restrictions

Not all Git server implementations support server side commit hooks. For example Gerrit (a Git server which also provides the ability to do code review) does not support hooks in this form. Also Github and Bitbucket do not support server hooks at the time of this writing.

Local hooks in the local repository can be removed by the developer.

Part XXIV. Good practices in Git

Chapter 63. Writing meaningful commit messages

63.1. Importance of Git commit messages

A *commit* adds a new version to the repository. This version is described by a *commit message*.

The *commit message* describes the changes recorded in a commit and helps the user to understand the history of the files contained in a Git repository.

A commit message should therefore be descriptive and informative without repeating the code changes.

63.2. Guidelines for useful commit messages

A commit message should have a header and a body. The header should be less than 50 characters and the body should wrap its text at 72 so that the commit message is displayed well on the command line or in graphical tools displaying the history. The body should be separated from the header by an empty line.

The body should mainly describe the reason why the change was made. The changes in the file can be reviewed with the help of Git.

The commit message should be in present tense, e.g., "Adds better error handling" instead of "Added better error handling".

The last paragraph can also contain *metadata* as key-value pairs, also referred to as the *commit message footer*. This metadata can be used to trigger a certain behavior. For example the *Gerrit* code review system uses the `Change-Id` key followed by a *change-id*, which does not change across different versions of the same code review. This *changed id* is used to identify to which review the message belongs.

The *commit message footer* can also have e.g., 'Signed-off-by' and may be used to link to a bug tracking system, e.g., 'Bug: 1234'.

63.3. Example message

The following can serve as an example for a commit message.

```
Short summary (less than 50 characters)

Detailed explanation, if required, line break at around 72 characters
more stuff to describe...

Fixes: bug #8009
Change-Id: I26b5f96ccb7b2293dc9b7a5cba0760294afba9fd
```

63.4. Example histories

The following listing shows the output of the `git log --oneline` command of a Git repository with bad commit messages. The first value in each line is the shortened SHA-1, the second the commit message. This history is not useful.

```
21a8456 update
29f4219 update
016c696 update
29bc541 update
740a130 initial commit
```

The next listing shows the history of another Git repository in which better commit messages have been used. This history already gives a good overview about the activities.

```
7455823 Bug 391086 - Search and filter the model editor tree.
9a84a8a Bug 404207 - Missing DynamicMenuContribution in child selector
952e014 Bug 404187 - Spelling error in Toolbar/Add child
71eeea9 Bug 402875 - Importing model elements from legacy RCP
123672c Bug 403679 - New Application wizard is missing dependencies
97cdb9a Bug 388635 - Creates an id for handlers
```

The above example also adds the corresponding bug number to the commit message. Some teams (like the Eclipse platform team) use this approach, others prefer to add the bug number to the commit messages.

Chapter 64. Workflows using separate repositories

64.1. Providing a patch

Git emphasizes the creation of branches for feature development or to create bug fixes. The following description lists a typical Git workflow for fixing a bug in your source code (files) and providing a patch for it. This patch contains the changes and can be used by another person to apply the changes to his local Git repository.

This description assumes that the developer who creates the changes cannot push changes directly to the remote repository. For example you solve an issue in the source code of an open source project and want that the maintainer of the project to integrate this change into this repository.

1. Clone the repository, in case you have not done that.
2. Create a new branch for the bug fix
3. Modify the files (source code)
4. Commit changes to your branch
5. Create patch
6. Send patch to another person or attach it to a bug report, so that is can be applied to the other Git repository

You may also want to commit several times during 3. and 4. and rebase your commits afterwards.

64.2. Working with two repositories

Sometimes you want to add a second remote repository to your local Git repository and pull from and push to both repositories. The following example describes how to add another remote repository and exchange commits with both repositories.

You can add another remote repository called *remote_name* via the following command.

```
# add remote
# syntax: git remote add <remote_name> <url_of_gitrepo>
# git remote add mysecondrepo <url_of_gitrepo>

git remote -v
# see all repos
```

For merging the changes in *mysecondrepo* create a new branch called *newbranch*.

```
# create a new branch which will be used
# to merge changes coming from repository 1
git checkout -b <newbranch>
```

Afterwards you can pull from your new repository called *mysecondrepo* and push to your original repository, e.g., origin.

```
# reminder: your active branch is newbranch

# pull remote_name and merge
git pull mysecondrepo

# or fetch and merge in two steps
git fetch mysecondrepo
git merge mysecondrepo/newbranch

# afterwards push to first repository
# -u sets the tracking branch
# for the current branch
git push -u origin master
```

64.3. Using pull requests

Another very common Git workflow is the usage of pull requests. In this workflow a developer clones a repository and once he thinks he has something useful for another clone or the origin repository he sends the owner a pull request asking to merge his changes.

A pull request can be seen as a notification which includes the information from which branch and URL the changes can be pulled and also the information to which URL and branch these changes should be pulled too.

This workflow is actively promoted by the GitHub.com hosting platform but you can also provide the required information to someone via email.

> **Tip**
>
> You can use the `git request-pull` command to generate a generic pull request which you may include into an email. See Git request-pull for details.

Chapter 65. Workflows using one shared central repository

65.1. A shared repository between developers

A very typical Git workflow is that the developers integrate their work via a shared remote repository. The following section describes a typical Git workflow for this scenario.

The shared repository is located on a server so that it can easily be reached by each developer.

The developers push to this remote repository, typically they use the remote `master` branch on the remote repository to integrate their work. They may also use different remote branches for shared feature development or maintenance releases.

The initial setup requires that every developer clones the remote repository or adds the remote repository as additional remote to his local repository.

65.2. Workflow

To develop a change and integrate it into the shared repository, the developer would:

1. Create a new local branch for the development
2. Change content in the working tree and add and commit his changes
3. If required he switches to other branches to do other work
4. Once the development in the branch is complete he rebases (or merges) the commit history onto the relevant remote-tracking branch to allow a fast-forward merge for this development
5. Pushes his changes to the remote repository; this results in a fast-forward merge in the remote repository

Note

Git emphasizes the creation of branches for feature development or to create bug fixes.

During this development he may fetch and merge or rebase the changes from the remote repository at any point in time. The developer may use the `pull` command instead of the `fetch` command.

65.3. Using branches

Even if you have the rights to push to master in a remote repository, creating a local branch for every feature or bug fix is a good practice.

Once your development is finished you merge your changes to your master and push the changes from master to the shared remote Git repository.

Index

Symbols

A

B

C