Manage Your Code with Git and Iceberg

Guillermo Polito and Stéphane Ducasse with Allex Oliveira

May 16, 2024

Layout and typography based on the `sbabook` LaTeX class by Damien Pollet.

Contents

Illustrations

CHAPTER 1

Preamble

Git is the defacto standard distributed source versioning system. Even though Pharo got its own distributed versioning system during more than 12 years. It lacked the maturity of git and one key features: branches. This is why since Pharo 6.0 it was clear that Pharo needed to have support for git. Pharo started to support git since Pharo 6.0. Pharo 7.0 saw a major version since all its code and development migrated to git.

However managing Pharo with git is not just a matter of saving code into files and versioning. Any fool can do that in one afternoon. Pharo has a powerful reflective layer and execution change the objects that represent code itself. Therefore Pharo as a living system can be in a different state than the file checkouted from a git repository. From this we can imagine many complex scenarios that even smart programmers would have headaches to understand.

Therefore the Pharo consortium (E. Lorenzano, N. Passerini, G. Polito and P. Tesone) developed an advanced tool to help managing the live programming aspect of Pharo and the static perspective that file-based versioning systems such git have of the reality. Iceberg not only supports the management of large projects (such as Pharo itself with more than 600 packages and a couple of thousand classes) but it helps us (the developers) to understand the situation between our image, the files on our disc and the multiple branches and remote repositories that the git model offers. It offers strategies to address problems we may face.

Iceberg is a tool to manage git projects from Pharo. It makes the experience of managing code really smooth. A major effort went into the version of Iceberg present in Pharo 7.0. We are using the version of Iceberg available in Pharo 8.0.

This document is under writing but we decided to release before its completion because managing code can be a large topic when we start to discuss workflow and process.

The book is structured for now in two main parts

1. Understanding from the command-line
2. Managing Pharo code with Iceberg.

The authors want to thank Sean de Nigris, Quentin Ducasse and Stefan Eggermont for the reviews and copy-edit of the early version. We also thank Peter Uhnak for his first blog on publishing Pharo code on Github. We thank Iona Thomas for the enhancements of Iceberg.

CHAPTER 2

Getting Started with Git

In this chapter we introduce the basics of `git` and Version Control Systems (VCSs) through guided examples. We first start by setting up a repository in a remote server and then load it in our own machine. We then show how we can inspect the state of our repository and save our files into it. Once our changes are saved, we show how we can push our changes to our remote repository in a distant server.

This chapter will assume you have `git` already installed in your machine, and that you're using a *nix operating system.

Moreover, you will see that we will approach `git` with the *command-line*. Don't be afraid if you've never used it before, it is not as difficult as it may seem and you will get used to it. There is always a first time! Also, we promise you that everything you learn in here can be applied to, and will actually help you better understand, non command-line tools.

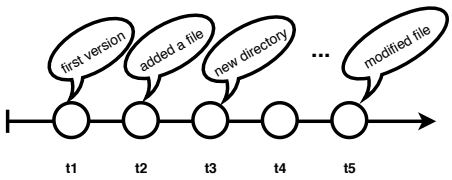

Figure 2-1 A repository as a timeline of changes.

2.1 Creating a Repository

A `git` repository is a store of files and directories. The big difference between a `git` repository and a simple filesystem is that the changes we make are stored as events, like a timeline (Figure 2-1). Git not only stores such a timeline but also allows us to query it, undo some of its changes, and so on.

Before working with such a history/repository we need to set it up. Fortunately, setting up a `git` repository is much lightweight than setting up a database repository. Though there are many different ways to create a `git` repository, we will start with a simple solution to be up and running as fast as possible. We will study other ways to setup repositories in Chapter 3.

Let's proceed to create a repository in an online hosting service such as GitHub or GitLab. On GitHub use the *New Repository* action. Figure 2-2 shows the kind of form GitHub provides to its users to create a new repository.

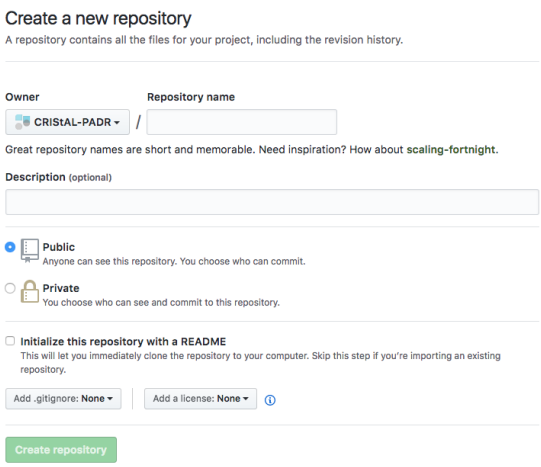

Figure 2-2 Creating a new repository on Github.

Following the form/wizard will eventually get you a running repository online. You will be most probably then redirected to your repository page. Figure 2-3 shows how such a page looks like in GitHub.

We are almost set to work from the command line now. However, we need to set-up our mind around a couple of extra concepts. The repository we just created does not exist in our machine. Actually, it is stored in some server maintained by GitHub/GitLab. To interact with this repository we will need a network connection. We will call this repository, living in a remote server, a **remote repository**.

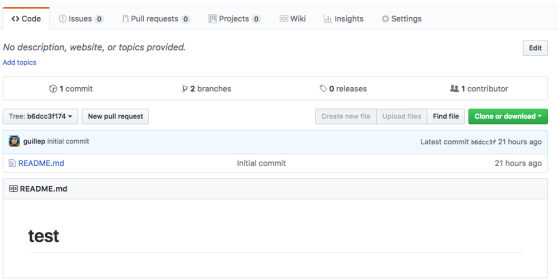

Figure 2-3 Repository page for a project called **test** in GitHub.

2.2 git clone

Git, constrastingly to other VCSs, is a distributed VCS. This has a lot of consequences in the way we work, that we will study in detail in Chapter 3. For now, you will have to remember only one thing: instead of being connected all the time to our remote repository, we will work on the repository on your machine (called a local repository). Eventually, we will synchronize the state between our **local** repository with the **remote** one (Figure 2-4). This is what makes possible the disconnected or off-line workflow that people often praise in `git`.

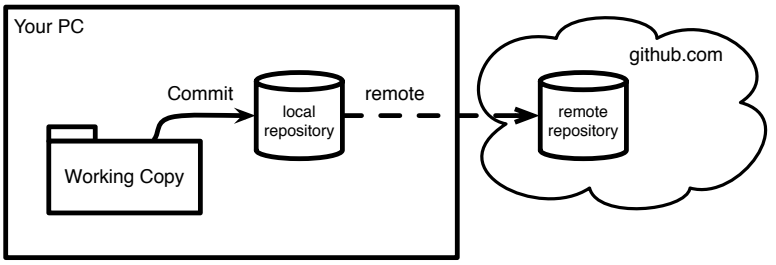

Figure 2-4 Basic git architecture: You change the files in your working copy, commit changes to local repository and synchronize your local repository with remote ones.

Making a local copy of a remote repository is such a common task that `git` has a dedicated command for it, the `git clone [url]` command. The `git clone` command receives as argument the URL of our repository, that we can get from our repository page. You will see that your repository page will offer you different URL options, the most used being SSH and HTTPS urls. We will use in this chapter HTTPS URLs because they have an easier setup, but for those readers that are curious, Section **??** compares SSH and HTTPS, and Section 4.7 shows how to setup your SSH environment.

To obtain the HTTPS URL of your repository, go to your repository's page and look for it under the clone/HTTPS options. As an example, Figure 2-5 illustrates how to get such url from a GitHub project page.

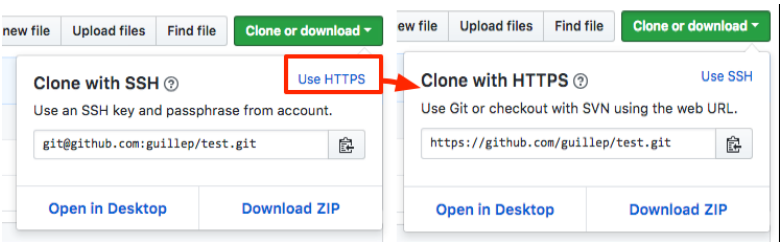

Figure 2-5 Getting the HTTPS url of your repository from GitHub.

Copy that url and use type your command as in:

```
$ git  clone [url]
Cloning into '[your_project_name]'...
remote: Counting objects: 11082, done.
remote: Compressing objects: 100% (4/4), done.
remote: Total 11082 (delta 2), reused 6 (delta 2), pack-reused 11076
Receiving objects: 100% (11082/11082), 4.35 MiB | 1.22 MiB/s, done.
Resolving deltas: 100% (4063/4063), done.
Checking connectivity... done.
```

When the command finishes, `git` is done creating a directory named as your repository (your_project_name). We will call this directory the **working directory** as it is where we will work and interact with our repository. The working directory, with all the files and directories it contains, is managed by `git` and linked to our repository. Do not worry, there is nothing else you need to do to keep this link, `git` will automatically track your changes for you.

You are now ready to go and start working on your project.

2.3 Making changes: How does `git` track my changes?

Let's now dive in our working directory and start making changes to it. We'll for example create a file called `project.txt`, then open it with a text editor and add some lines to it.

```
$ cd your_project_name
$ touch project.txt
...
```

After some time working, we can use the `ls` command to check the files in my directory.

```
$ ls
project.txt README.md
```

And then the `cat` command to check their contents from the command line.

```
$ cat project.txt
# Done
 - created the repository
 - ==git== clone
 - created this file

# To do
 - commit this file
 - push it to my remote repository
```

```
$ cat README.md
#My Project

This project is an example ==git== repository used to learn ==git==.
Check the project.txt file for information about pending tasks.
```

Basically, we have modified some files, but we have done no `git` at all. What does `git` know about these files at this point?

`git` does Nothing without your Permission

A new important thing to grasp about `git` at this point is that it will do nothing until we explicitly ask it to do it. In this sense, `git` is not any kind of repository but a transactional repository. All changes we do in our working directory are not stored by `git` automatically. Instead, we need to explicitly store them using a `commit` command, as we do with transactional database to store any data.

The transactional aspect means also that:

- while we do not commit, we can easily rollback our changes;
- the other side of the coin, until we commit all our changes are in a transient state, and we may lose them.

We will see more of this *explicitness* applied in other cases in the course of this book. Sometimes it may seem that `git` is just dumb, and that it cannot guess what we want to do when it is obvious. However, the case is that `git` has so many possibilities that not guessing is the healthier decision in most cases. Especially when considering destructive operations that may make you lose hours of work.

`git status`

We can then turn our question around: Does `git` know something about these files? Git indeed tracks our files to know what should be saved and what

should not. We can use this information to see what are actually the changes that happened while we were working. The `git status` command produces a list of the current changes.

```
$ git status
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:   README.md

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

  project.txt

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

Reading the output of `git status` we can see that it lists several information:

- **Changes not staged for commit.** Lists the modified files that `git` already knows.
- **Untracked Files.** Lists the files that are in the working directory but were never added under the control of `git`.

Also, `git status` shows some hints about possible commands that we may use next such as `git add` or `git checkout`.

`git` and directories

You may have observed a curious behaviour with directories. If you create an empty directory and then use the `git status` command, you'll notice the directory is not listed at all. It even looks like the directory is being completely ignored. For example, if we try adding an empty directory into a new repository `git` will actually say *working tree clean* as if there were no changes at all.

```
$ mkdir emptyDirectory
$ git status
On branch master
nothing to commit, working tree clean
```

Indeed, `git` does not manage empty directories but only files. Directories are only modelled as paths to get to files. No extra information is tracked for them. In other words, we cannot just store directories into `git`. And in case we want to do it for some reason, we need to put files into them.

2.4 Commiting your changes

We would like now to save our changes in our `git` repository. This way, if anything happens, we can always recover our work up to this point. We have said before that the operation of saving our work in the repository is called a **commit**. If we try the `git commit` command we will see this is not as direct as expected.

```
$git commit
On branch master

Initial commit

Untracked files:
  README.md
  project.txt

nothing added to commit but untracked files present
```

If we read Git's message, we will notice that though it has correctly identified that we have new files, `git` is asking us to *add* them before it can commit them.

The Groceries Metaphore

To make it simple, you can see this whole tracking story as going to the supermarket. Imagine you make your grocery list and go to the supermarket. To get our groceries and take them home, we need first to go look for them, put them in our shopping cart, and then go and pay for them. An extra service may propose to take your grocery list and do the groceries for you. But such an extra service requires that you make a list up-front.

Same apply with `git`, the default behavior of `git` is not to commit all the changes. Partly because Git's philosophy is to ask the user explicitly what to do, which in this case is translated to asking what to commit. Instead, `git` requires us to add the files we want to commit to a list of *added* files, also called in `git` terminology the *staging area*, and equivalent to your shopping list. Once our staging area is full with the changes we want to commit, we can commit such changes using the `git commit` command.

A first commit

Adding changes to your staging area is done through the `git add [file]` command. Let's proceed to add our changes and see what is the status of our repository afterwards.

```
$ git  add README.md
$ git  add project.txt
$ git  status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

  modified:   README.md
  new file:   project.txt
```

Now `git` says that our two new files are listed as "to be committed". Let's now proceed to save our changes in the repository with the `git commit -m "[message]"` command. The *message* used as argument of this command is a piece of text that we can use to explain the contents of the changes, or the intention of our changes.

```
$ git  commit -m "first version"
[master a93c016] first version
 2 files changed, 12 insertions(+), 1 deletion(-)
 create mode 100644 project.txt
```

If we check the status of our repository after the commit is done, we see that it has changed. There is nothing to commit:

```
$ git  status
On branch master
nothing to commit, working directory clean
```

Add then commit, all over again

If we repeat the process and we change one of our existing files, we will see something interesting. Commiting our changes in a file we added before requires that we do a `git add [file]` and `git commit` again on the same file, even if `git` already knew about it.

```
$ cat project.txt
# Done
 - created the repository
 - ==git== clone
 - created this file
 - commit this file

# To do
 - push it to my remote repository
```

```
$ git  status
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:   project.txt

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

This is because even if on the surface `git` seems to manage files, it actually manages *changes* to those files. Technically speaking, the changes we have done are *new* changes, so we have to tell `git` we are interested in those changes.

```
$ git  add project.txt
$ git  commit -m "Commit is not in ToDo anymore"
[master e14a09f] Commit is not in ToDo anymore
 1 file changed, 1 insertion(+), 1 deletion(-)
```

2.5 Synchronizing with your Remote Repository

So far we have worked only on the local repository residing in our machine. This means that mostly all of `git` features are available without requiring any internet connection, making it suitable for working off-line (think on working on the train or with a constrained connection!). However, working off-line is a two-edged sword: all your changes are also captive in your machine. While your changes are in your machine, nobody else can contribute or collaborate to them. Moreover, losing your machine would mean losing all your changes too. Keeping your changes safe means to synchronize them from time to time with your remote repository.

`git`'s metaphore for remote synchronization is based on the ideas of *pulling* and *pushing* changes between repositories.`git` takes the perspective that we are located in our local repository. We bring other's changes by *pulling* them from remote repositories to our local repository. We send our changes by *pushing* them from our local repositories to one or many remote repositories.

Getting Remote Changes with `git pull`

Before being able to share our commits in some external server, we need before to update our repository to avoid them being de-synchronized. While you can always try to share your commits by directly pushing (see Section 3.6), you will see with experience that `git` favors pulling before pushing. This is, among others, because in your local repository you have complete control to do whatever manipulation you want, what is especially important to solve mistakes and merge conflicts. You cannot do the same in your remote repository.

In our example pulling does not seem really necessary because you are the only person modifying your repository. No new changes happened in the remote repository in the meantime. However, let's imagine that you have done

a modification in this same repository from another machine or even a different clone in the same machine (which are totally feasible scenarii). In that case, you would like to update your local repository with those new changes.

Updating our repository is done through the `git pull` command. Pulling will update our database and then update our files.

```
$ git  pull
remote: Counting objects: 2, done.
remote: Compressing objects: 100% (1/1), done.
remote: Total 2 (delta 1), reused 2 (delta 1), pack-reused 0
Unpacking objects: 100% (2/2), done.
From https://github.com/guillep/test
   1656797..a2dbd8b  master     -> origin/master
Updating 1656797..a2dbd8b
Fast-forward
 newfile | 0
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 newfile
```

A Bit on Merging

We will see in detail in Section 3.6 that `git pull` performs two different operations: a fetch and a merge. The fetch lookups new commits in remote repositories. The merge, studied in detail in Section 3.3, takes the state in the remote repository and your local repository and tries to make a single version of of that. Three different scenarii can actually happen from a pull operation, which will be:

- **Fast-forward**: the updates were applied without needing a merge.
- **Automatic Merge**: the updates were applied without conflicts. `git` had to do a merge commit and will ask you for a commit message.
- **Merge Conflict**: The changes you did and incoming changes affect some common files. In this case `git` does not know what version to keep (or even if a mixture is possible) and asks you to solve it manually before doing a new commit.

Once the merge is resolved, your working copy is updated with the new version of your repository. Luckily for us, fast-forward and automatic merges are the simplest and more common ones. They require almost no manual interaction other than introducing a message.

Sending your Changes with `git push`

The final step in our `git` journey is to share our changes to the world. Such sharing is done by **push**ing commits to a remote repository, as shown in Figure 3-13. To push, you need to use the `git` command `git push [remote]`

[remote_branch]. This command will send the commits pointed from your your current branch to the remote [remote] in the branch [remote_branch].

```
$ git  push origin master
Counting objects: 3, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 271 bytes | 0 bytes/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To git@github.com:[your_username]/[your_repo_name].git
   b6dcc3f..f269295  master -> temp
```

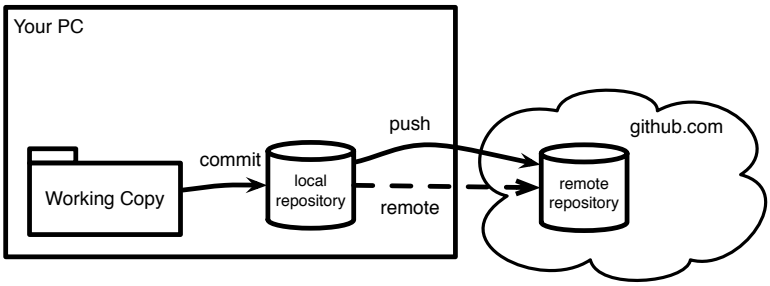

Figure 2-6 Push is an operation that sends commits from your local repository to a remote repository.

A Branch's Upstream

We can omit the destination branch and remote from the command, relying on `git` default values. By default a `git push` operation will push to the so called **branch's upstream**. A branch's upstream is a configuration specifying a pair (remote, branch) where we should push by default that branch. When we clone a repository, the default branch comes with an already configured upstream. We can interrogate `git` for the branch's upstream with the super verbose flag in the branch command, *i.e.*, `git branch -vv`, where we can see for example that our **master** branch's upstream is **origin/master**, while our **development** branch has no upstream.

```
$ git  branch -vv   # doubly verbose!
  development   1656797 This commit adds a new feature
  master        f269295 [origin/master] First commit
```

When a branch has no upstream, a push operation will by default fail with a `git` error. Git will ask us to set an upstream, or otherwise specify explicitly a pair remote/branch for each push.

```
$ git  push
fatal: The current branch test has no upstream branch.
To push the current branch and set the remote as upstream, use

    ==git== push --set-upstream origin test

$ git  push --set-upstream origin test
Counting objects: 3, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 271 bytes | 0 bytes/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To git@github.com:[your_username]/[your_repo_name].git
   b6dcc3f..f269295  master -> test
```

Pushes can get Rejected

In some scenarii `git` may reject our pushes, so they are not saved to the remote repository. In general `git` rejects changes when the remote repository has diverged from ours. Of course a rejection may also happen when we don't have write permissions in the remote repository. The typical error shows something like the following:

```
$ git  push
To git@github.com:guillep/test.git
 ! [rejected]        master -> master (fetch first)
error: failed to push some refs to
    'git@github.com:[your_username]/[your_repo_name].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.
```

As the error message says, the remote has changes that we do not have locally. In other words, our push has been rejected because otherwise we would have overwritten the remote changes. Instead, we need to take the remote changes and *mix and match* them with our changes, by applying a pull (Section **??**) and a merge (3.3). After the pull, our repository will have our outgoing changes, but no more incoming changes, and so our push will not be rejected

2.6 Overview

In this chapter we have studied the basic operations of `git`. We have seen that a new repository starts in your local machine with a `git clone` that creates a working copy directory. Changes in our working copy are tracked by `git` automatically and we can query the tracking using `git status`. We can then proceed to operate on our changes as illustrated in 2-7.

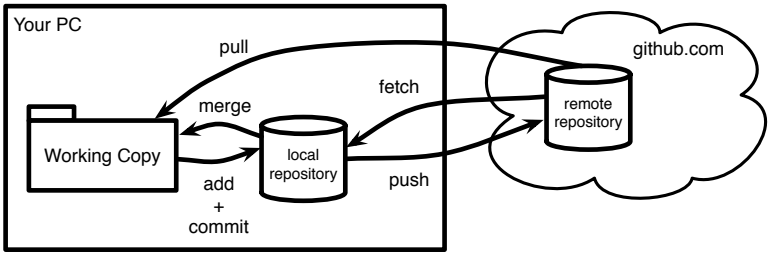

Figure 2-7 Overview of `git` basic operations: `add`, `commit`, `pull` and `push` (+ extra `fetch` and `merge`).

We have seen that:

- `git add` tells `git` about files to track for commit.
- `git commit` finishes a transaction and stores our changes in a commit.
- `git pull` synchronizes a remote repository with our local repository by doing first a `fetch` and then a `merge`. Different merge scenarios may happen, in which some of them cause conflicts that have to be manually resolved.
- `git push` sends our changes to a remote repository. A push can get rejected.

2.7 Exercises

1. **Exercise 1**. Create an account in your preferred `git` repository hosting service, create there a repository and clone it. Then, check its history from the command line. How many commits are there in the repository? Tip: there is a difference if you checked the "create README.md file" checkbox while creating a repository.
2. **Exercise 2**. Create a file, a file inside a directory and an empty directory. Commit them (remember, `git add`, `git commit`). What can you see there? How does `git` manage directories?

3. **Exercise 3**. If you're on a unix system (linux/osx), try changing your file's permissions and check `git status`. How does `git` treat file permissions? Commit your changes and check the log. What can you observe?
4. **Exercise 4**. Push now your changes to your remote repository. Then, clone your repository again in another directory. Tip: try checking the help of the clone command `git clone -h`. Did `git` save all your files, directories and even permissions?
5. **Exercise 5**. Go back to your first repository, add a new file, commit it and push it. Then go back to the second repository and pull. Inspect the history in both repositories: Is it the same?
6. **Exercise 6**. Check your online repository on your hosting service. Can you see the same state as in your local repositories? Go over the different tools offered by the hosting, they usually give some idea of the activity of the project, try to understand what they are for.

CHAPTER 3

Understanding Git

Before going on with the reproducibility concerns that brought you here to read this chapter and even before continuing with practical `git` commands, we will dive a bit into `git` concepts.

3.1 Some `git` Internals

Understanding a bit how `git` works is useful when doing some more complicated stuff such as merging and branching. If you already know what is a `git` commit, a `git` reference and how the graph of `git` objects is managed, you can skip this section.

Dissecting a `git` Repository

Before starting explaining what is a commit, what is a branch, and so on, let's start easy by understanding the parts that compose our `git` repository. When you create a `git` repository as we did in the last section, or you clone an old repository that already has some files in it, you will find that there is more than meets the eye. A `git` repository has usually three core collaborating components: the **working copy**, the **repository**, and the **remote** repositories. You can see an schematics on Figure 3-1.

What you usually see in your disk when you clone is not actually the `git` repository but the **working copy**. The working copy is the directory where your files are, where you work and apply modifications. It is called a working **copy** because what you see is actually a copy of what is in the repository. The working copy is a write-able copy: you can freely modify it, break it, add new things or remove them.

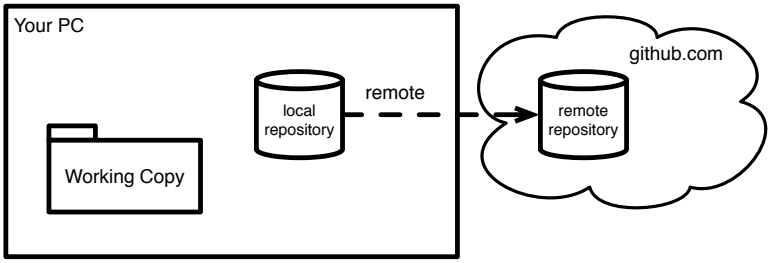

Figure 3-1 `git` repository structure: the working copy, the repository, and the remote repositories.

Actually, you can do whatever change you want in your working copy: `git` will not take it into account, at least not automatically. Once your changes are ready, you have to commit them into your repository to store them in your repository. A commit will take your changes, freeze them, and store them in the local database. Just for the curious ones, the local database (also known as **the BLOB** in the `git` jargon) is stored inside your working copy, in a hidden directory called **.git**.

The commits you create from your changes live only inside your machine by default. If you want to share your commits with others, or to import commits from some fellow colleague, you have to interact with a remote repository (also called just **remote**). A remote is a distant `git` repository that you will synchronize with your local one from time to time. This is where the famous pull and push come into play!.

Of course, this is an utterly simplified scenario. You could have a repository without a working copy. And your repository may have many remotes to synchronize with. But we will get into more complex stuff early on, no need to rush now.

A history-aware transactional database?

As we explained before, we usually work on the working copy, modifying our files and directories. Once we finished some work, we can freeze it and store it in the repository. That's what we call a **commit**.

From this perspective, a `git` repository works as a transactional database. You are working on the changes of your disk, but they will not be effectively applied until you finish your transaction. Doing your transaction is performed, as in the database world, using the **commit** command. The result of this transaction is to create a new commit object in the `git` repository. This commit object will contain an id (usually a hash such as `7ba52e5`) plus all changes we wanted to apply.

git will store your last changes but also remember the entire history of changes you did. It keeps a list of all changes you did so you can do some nice stuff like for example:

- come back in time to recover some old changes,
- trace the changes in a file to see who (and why!) did a change, or
- analyze your repository and do some archeology, to see how your project evolved.

It's a just graph of commits

The history of commits we explained before is not stored in a list form but in a graph form. A commit is a node connected to other commits by **parenthood**. A commit is said to be parent of another commit if it is the exact previous version. In other words, when we create a new commit, the parent of our new commit is the previous commit. A commit is said to be an ancestor of another commit if it preceeds it in history. Moreover, a commit can have one or many parents, and many commits can have the same commit as parent.

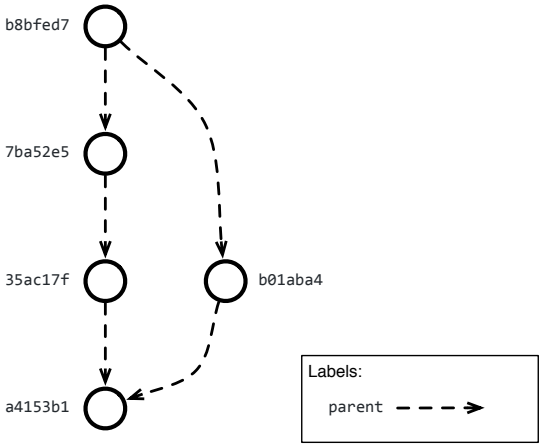

Figure 3-2 Graph of commits.

For instance, take a look at the schema of a typical commit graph represented in Figure 3-2.

- Commit a4153b1 is the first commit in the graph, with no parents. A commit with no parents represents the first commit in a repository, when no previous history was available.
- Commit 35ac17f's parent is a4153b1 and commit 7ba52e5's parent is 35ac17f.

- Commit `b01aba4`'s parent is also `a4153b1`.
- Commit `b8bfed7` has two parents: `7ba52e5` and `b01aba4`.

You may be asking yourself how can we arrive to such a situation. In short, a commit that is parent of many commits is creating an alternative history line: it is the result of a **branch** operation. Likewise, a commit that has many parents is joining two histories: it is the result of a **merge** operation.

Naming commits with references

You probably noticed that referring to commits by their id is awkward. Commit ids are generated automatically as hashes that avoid duplications as much as possible. However, they are not handy to work on a daily basis since they are hard to remember and type.

To solve this, `git` provides a second kind of objects: `git` **references**. A `git` reference is like a label that you put on a commit, to be able to identify that commit by a much simpler name afterwards. For example, you can name a commit as **release 1.0** or you can name it as **current development commit**.

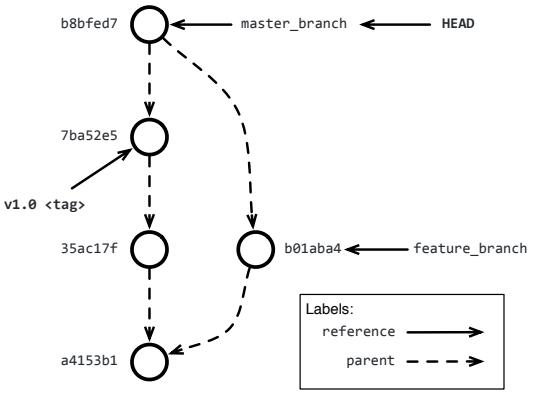

Figure 3-3 `git` references: a reference refers to a commit. HEAD to a branch and tags are special fixed references.

As we show in Figure 3-3, there are two main kinds of references in Git:

- **tags**: tags are fixed labels that once created are not meant to be removed or moved. They are useful for doing releases: people will expect that a release does not change, otherwise they cannot depend on it.
- **branches**: branches are transferable labels that can be moved from commit to commit. They are used to maintain the different history lines of your project.

Another special reference, called **HEAD** is internally used by `git` to know what is our current working branch. **HEAD** usually refers to a branch, not directly a commit. While it would look like an implementation detail, knowing that **HEAD** is there can save you many headaches as we will see later.

What you should see is that master, development are just references too.

Now that you have built some strong conceptual `git` muscles, we can continue in the next sections with some practical Git. Do not hesitate to come back to these sections to refresh some of the basics. As with any sport or discipline, understanding and practicing the basics is really important, since everything else is based on them.

3.2 Understanding Detached HEAD

When your project is in a stable state, it is often good to freeze it and put a name to that version. That way, other users can load the frozen version using that well-known name, and also be sure that version will not change. Freezing a version is particularly useful to reproduce a piece of sofware. A frozen version can be reloaded exactly as it is right now but in some point in the future. Thus, software that depends on a frozen version can also benefit from its stability.

In `git`, releasing is done via tags. A tag is a label that we put on a particular commit to be able to find it easily later on, so remember to put short, readable names to them. One particular consideration about tags is that they are not meant to be modified, although you will find in Git's documentation that you have special operations (that we do not recommend) to do that.

To create a tag, use the command `git tag` giving as argument a name for the tag and a descriptive message. Usual tag names use semantic version conventions, prefixed with a v. For example version 1 would be `v1.0.0`.

```
$ git tag -a v1.0.0 -m "First stable release"
```

You can afterwards list all your tags using the `git tag` command without arguments:

```
$ git tag
v0.1.1-alpha
v1.0.0
```

Finally, if you want to recover the code that you tagged at some point, you can use the `checkout` command with the name of your tag.

```
$ git checkout v1.0.0
Note: checking out 'v1.0.0'.

You are in 'detached HEAD' state. You can look around, make
    experimental changes and commit them, and you can discard any
```

```
    commits you make in this state without impacting any branches by
    performing another checkout.

If you want to create a new branch to retain commits you create, you
    may do so (now or later) by using -b with the checkout command
    again. Example:

  git checkout -b <new-branch-name>

HEAD is now at 0c0e5ff... Adding a title
```

When checking out a tag, `git` tells you that we are in *detached HEAD* state. And that whatever commit we do in this state will be lost unless we create a branch. What happened here is that the `checkout` command modified the **HEAD** reference to point to the *commit pointed by the tag, instead of a branch*. Figure 3-4 shows the commit graph for this particular case.

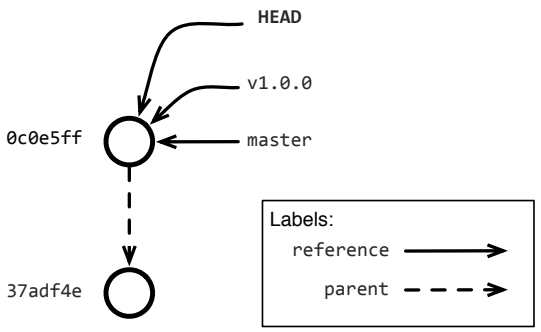

Figure 3-4 Detached HEAD after checking out a tag: HEAD refers to a commit and not a branch anymore.

When you are in a detached HEAD, if you want to save your modification, you should check out an existing branch using `git checkout .`

3.3 Merging history lines

The most complicated part of `git` is not branching or commiting, but merging. In our time-travel time-line metaphore we said that branching is equivalent to open new time-lines. Merging is the equivalent to join them into a single history.

The concept behind merging is not difficult. Using the same idea of graph of commits that we used before, a merge can be represented as a commit that has several parents, thus joining several histories. Figure 3-5 illustrates such a merge commit.

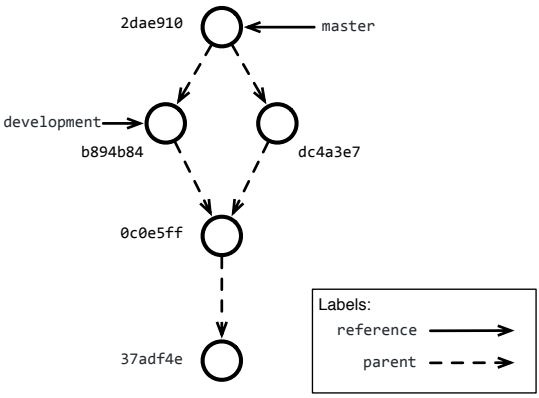

Figure 3-5 Merging the history with a merge commit.

However, as you see also in the picture, a merge commit will be referenced by one of the branches but not both. In other words, *a merge operation means that a first branch will be merged into a second one*. Thus the first one will remain intact. To perform a merge we need to checkout the branch that will host the changes, and then use the `merge` command with a branch name as argument. The following example shows how we can merge the development branch into the master branch.

```
$ git checkout master
 ...
$ git merge development
[Merge made by the 'recursive' strategy.
 ...
 1 file changed, 0 insertions(+), 0 deletions(-)
 ...]
```

Managing Conflicts

When merging different history lines, things can go wrong if both history lines modified the same file or ressource. Such a problem is also called a **conflict**.

To understand the issue, let's generate a conflict on purpose. We can create two branches called `future-1` and `future-2` adding each the same file but with different contents:

```
$ git checkout -b future-1
$ echo "I'm in future-1" > conflicting.txt
$ git add conflicting.txt
$ git commit -m "Maybe will cause a conflict"
```

```
# Let's go back to master and redo the same in another branch
$ git checkout master

$ git checkout -b future-2
$ echo "I'm in future-2" > conflicting.txt
$ git add conflicting.txt
$ git commit -m "I'm sure it will cause a conflict!"
```

And then trigger a conflict when trying to merge:

```
# We are in future-2 so we will try to merge future-1
$ git merge future-1
Auto-merging conflicting.txt
CONFLICT (add/add): Merge conflict in conflicting.txt
Automatic merge failed; fix conflicts and then commit the result.
```

We see that as soon as we merge, `git` tries to automatically merge the file `conflicting.txt`. It detects however a merge conflict that does not allow it to continue. If we check Git's status, you will now see:

```
$ git status
On branch future-2
You have unmerged paths.
  (fix conflicts and run "git commit")

Unmerged paths:
  (use "git add <file>..." to mark resolution)

  both added:      conflicting.txt

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

`git` tells us that `conflicting.txt` is not merged and that we should fix it. To continue working, we should resolve such a conflict, telling `git` what version we want to keep. Several solutions work: either we keep the version we had in **future-2**, we keep the version incoming from **future-1**, or we keep a can manually resolve the conflict and keep whatever version we want.

The easiest, non-thinking, way to merge is to open the conflicting file and resolve the conflict. For example, if we open our `conflicting.txt` file with a text editor we will see:

```
<<<<<<< HEAD
I'm in future-2
=======
I'm in future-1
>>>>>>> future-1
```

`git` modified our file adding some `<<<<<<<`, `>>>>>>>` and `=======` markers in our file. What this markers delimit is the conflicts `git` found. As the first line says, the first region (what is between the `<<<<<<<` and the `=======`)

corresponds at the version that was in **HEAD** (i.e., **future-2**). As the last line says, the last region (what is between the `=======` and the `>>>>>>>`) corresponds to the version that was in **future-1**.

To resolve the conflict, you should:

- remove all the special markers
- keep only the version you want (or edit it to be different)
- add and commit the conflicting file

For example, let's say we wanted to keep the version in **future-2**, we can edit the file leaving only

```
I'm in future-2
```

and then commit the resolved conflict:

```
$ git add conflicting.txt
$ git commit -m "Resolve conflict"
```

3.4 Commit in workflow

Commiting means that we are going to move some content from our working copy to our local repository, as it is shown in Figure 5-1.

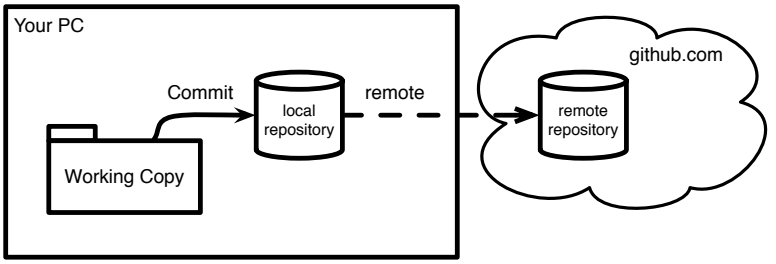

Figure 3-6 Commit is an operation that stores things from your working copy into your local repository.

What the commit command is doing behind is to create a new node in our history graph. Moreover, it will update the master branch label to point to this new commit. The commit graph in this case will look as in Figure 3-7.

If we repeat the process, i.e., we apply a change to one of our files, add and commit our commit graph will change again. A new commit with a new commit id will be created having as parent our previous commit. The master branch label will be updated and point to this new commit. The commit graph in this case will look as in Figure 3-8. Notice how our old commit is still there, but it's accessible as the parent of our new commit.

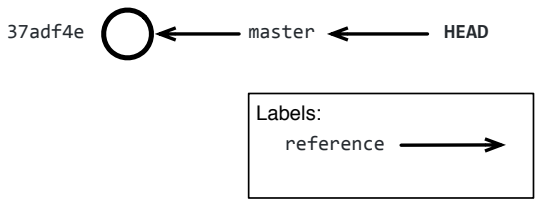

Figure 3-7 History graph after our first commit.

```
$ git add README.md
$ git commit -m "Adding a title"
[master 0c0e5ff] Adding a title
 1 file changed, 1 insertion(+)
```

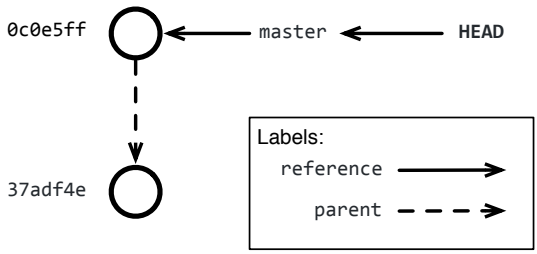

Figure 3-8 History graph after our second commit

3.5 Creating new history lines with branches

Branches in `git` represent different histories. As in one of science fiction time-travel theories, `git` branching is equivalent to take one moment in time have several alternative time-lines from there. Figure 3-9 illustrates the idea, showing that you can have two different futures from commit `0c0e5ff`.

By default, a `git` repository will include a single branch, called **master**. Most people only need a single branch to work. However, it may be useful to split work in several branches as we will see later. You can ask `git` for the branches in the repository using the command `git branch -v`.

```
$ git branch -v
* master  0c0e5ff Adding a title
```

This command shows all branches in the repository, one per line. Then, for

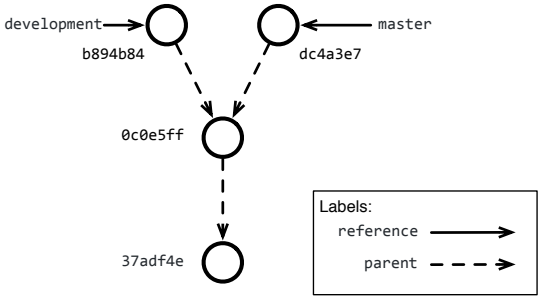

Figure 3-9 History lines can be branched from a commit.

each branch it shows what commit it points, and the comment on that commit.

Creating a new branch

To create a new branch, we can use the command `git branch [branch_name]` giving as argument the new branch name. This will create a new branch from our current commit, the one that can be resolved from `HEAD`. Figure 3-10 shows what happens in the graph view.

```
$ git branch development
```

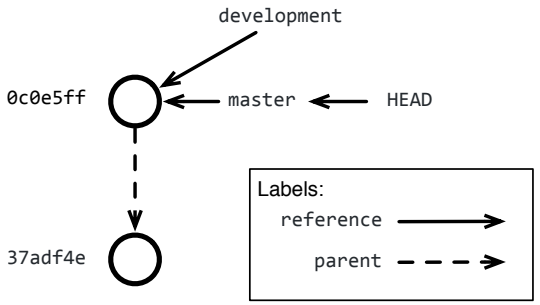

Figure 3-10 A new branch points by default to the same commit as the current branch.

However, as we see in the graph view, creating a new branch does not modify `HEAD`. Indeed, our current branch/commit did not move. We will observe the same in the command line, if we ask the list of branches. The branch master

is marked with a star, indicating it is the actual branch. And both branches point to the same commit.

```
$ git branch -v
* master      0c0e5ff Adding a title
  development 0c0e5ff Adding a title
```

To start working on our new branch, we just need to use the same `checkout` command we used for tags.

```
$ git checkout development
Switched to branch 'development'
```

Or alternatively, we could have created our branch using the `checkout -b` command, which performs a `git branch` and a `git checkout` one after the other. Useful since these operations are usually done together most of the time.

```
# Instead of branch and then checkout
$ git checkout -b development
Switched to branch 'development'
```

Then, doing some work and creating a commit will only modify our current branch and leave `master` as it was before.

```
$ touch somefile
$ git add somefile
$ git commit -m "added somefile"
[development b894b84] added somefile
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 somefile
$ git branch -v
   master      0c0e5ff Adding a title
 * development b894b84 added somefile
```

Diverging history

Now that we have done some work in a branch, we can make our branches diverge. We only need to checkout another branch, existing or new, and start working from there.

```
$ git checkout master
Switched to branch 'master'
$ touch someotherfile
$ git add someotherfile
$ git commit -m "added someotherfile"
[master dc4a3e7] added someotherfile
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 someotherfile
$ git branch -v
 * master      dc4a3e7 added someotherfile
   development b894b84 added somefile
```

This change will create two different history lines, as shown in Figure 3-11. One history line represented by the `master` branch, and another history line represented by the `development` branch.

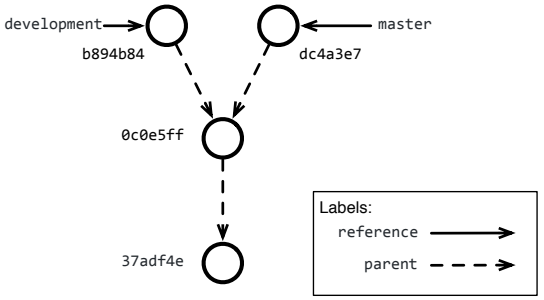

Figure 3-11 Divergent history.

3.6 Interacting with Remote Repositories

So far we have worked only on the repository that resides locally in our machine. This means that mostly all of `git` features are available without requiring an internet connection, making it suitable for working off-line. However, working off-line is a two-edged sword: all your changes are captive in your machine. While your changes are in your machine, nobody else can contribute or collaborate to them. Moreover, losing your machine would mean losing all your changes too.

Keeping your changes safe means to synchronize them from time to time with a **remote repository**. A remote repository is a copy of your local repository that is stored remotely, that is, in somewhere else's machine. This could be, for example, in your company's or university's server, the cloud, etc.

In this section we will see how to interact with remotes, how to configure them, and how to synchronize our local repository with them.

`git` Remotes

A `git` remote is a `git` server that is hosted in some machine other than ours. Usually, a remote will be hosted by some company like GitHub or GitLab, but it can be hosted also within our own company/university/research laboratory. Actually, we have already worked with a remote without knowing it, when we have cloned our repository in Section 2.2. The code we used in that moment was:

```
$ git clone git@github.com:[your_username]/[your_repo_name].git
```

Which can be generalized as:

```
$ git clone [remote]
```

Once created, we can interrogate our repository for its remotes using the command `git remote -v`. We will then observe that git created automatically a remote named **origin** pointing to the location that we just cloned.

```
$ git remote -v
origin  git@github.com:[your_username]/[your_repo_name].git (fetch)
origin  git@github.com:[your_username]/[your_repo_name].git (push)
```

This first means that `git` allows us to assign a name to avoid using urls all the way. In addition, we can see that `git` differentiates remotes used for **fetch**ing from those used for **push**ing. Those differences are important for more advanced git configuration, that we will not cover in this chapter.

Adding and Removing Remotes

For advanced scenarios, when we need more than the default **origin** remote, we will need to use different remotes. All git commands interacting with a remote repository will have a variant accepting a remote repository as argument, as we will see later. In those cases, we can specify the remote's url on each of those commands to interact with the desired remote.

However, to avoid copy-pasting different remote urls all the time, `git` provides us with the possibility of configuring new **named remotes** such as origin. The drawback of such an approach is that our list of remotes will need to be maintained from time to time, for example, if urls become invalid or our repository moves. In such cases, we will want to modify or remove old remotes to keep avoid errors or mistakes.

To create a new named remote we can execute the command `git remote add [remote_name] [url]`.

```
$ git remote add someRemote [url]
$ git remote -v
origin  git@github.com:[your_username]/[your_repo_name].git (fetch)
origin  git@github.com:[your_username]/[your_repo_name].git (push)
someRemote [url] (fetch)
someRemote [url] (push)
```

Existing remotes can then be renamed using the `git remote rename [old_name] [new_name]`. And in case the remote name you wanted to rename does not exist, `git` will answer you with a fatal error.

```
$ git remote rename someRemote company_remote
$ git remote rename non_existent newname
fatal: No such remote: non_existent
```

Existing remotes can then be renamed using the `git remote rename [old_name] [new_name]`. And in case the remote name you wanted to rename does not exist, `git` will answer you with a falta error.

```
$ git remote rename someRemote company_remote
$ git remote rename non_existent newname
fatal: No such remote: non_existent
```

Finally, to remove an existing **named remote** you can use the `git remote remove [remote_name]`. And in case the remote name you wanted to rename does not exist, `git` will answer you with a falta error.

```
$ git remote remove company_remote
$ git remote remove non_existent
fatal: No such remote: non_existent
```

Update your repository: Fetching and Pulling

Before being able to share our commits in some external server, we need before to update our repository to avoid them being out of synchronization. While you can always try to share your commits by pushing (see Section 3.6), you will see with experience that `git` favors pulling before pushing. This is, among others, because in your local repository you can do whatever manipulation you want to solve mistakes and merge conflicts, while you cannot do the same in your remote repository.

Concretely, when using `git` you have to have a state of mind where:

1. . you update your repository
2. . you fix **locally** whatever existing conflict between your work and the remote work
3. . you then publish your changes.

Actually, our recommended workflow has one more step before updating: commit. If you try to update when your working copy is dirty, updating can destroy your changes. Instead, if you commit before doing an update, your changes will be safely stored in the database. You'll be able to do any expert manipulation with your changes once they are in the repository. As we said before, a `git` repository is no other than a database. It is a database that stores commits and references to those commits. And to update this database, we require two basic operations:

- **fetch**. Bring the commits and references from a remote repository to your local repository without affecting your own.
- **merge**. Merge the remote references with your own references, the same operation explained in Section 3.3.

In addition, the **pull** operation does both fetch and merge in a single operation (Figure 3-12).

Fetching is done through the `git fetch [remote]` command, where we can specify both a remote url or a remote name as remote. Or, if we don't specify a remote, `git` will by default fetch from whatever remote is specified as **origin**. Executing a **fetch** will show an output like the following:

```
$ git fetch [remote_name]
remote: Counting objects: 79, done.
remote: Compressing objects: 100% (23/23), done.
remote: Total 79 (delta 52), reused 74 (delta 52), pack-reused 4
Unpacking objects: 100% (79/79), done.
From git://github.com/[project_owner]/[your_repo_name]
   6b52ae6..5c53245  development -> [remote_name]/development
 * [new branch]      issue/876 -> [remote_name]/issue/876
 * [new tag]         v1.0     -> v1.0
```

Indeed, fetch will bring some objects (e.g., commits) to our repository, bring new branches and so on, but it will not update any of your branches or tags. We can then procceed to merge our local branch with the one in the remote by doing a normal merge operation but indicating a **remote branch** (that is, a branch prefixed by its remote name). Of course, as any merge operation, this can incurr into a conflict, that we should fix locally before continuing.

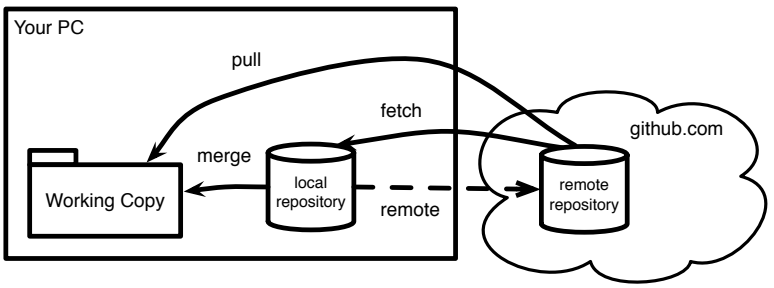

Figure 3-12 Fetch is an operation that brings things from a remote into your local repository. Merge will join the remote history with your current history and update your working copy. Pull will do both of them.

```
$ git merge [remote_name]/master
[Merge made by the 'recursive' strategy.
 ...
 1 file changed, 10 insertions(+), 1 deletions(-)
 ...]
```

These both operations could have been replaced by a `git pull [remote_name] [branch_name]` command. Pulling will fetch all commits from the branch named [branch_name] in the remote [remote_name] and then merge those commits with your current branch.

Share your commits: Pushing

The final step in our `git` journey is to share our changes to the world. Such sharing is done by **push**ing commits to a remote repository, as shown in Figure 3-13. To push, you need to use the git command `git push [remote] [remote_branch]`. This command will send the commits pointed from your your current branch to the remote [remote] in the branch [remote_branch].

```
$ git push origin temp
Counting objects: 3, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 271 bytes | 0 bytes/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To git@github.com:[your_username]/[your_repo_name].git
   b6dcc3f..f269295  master -> temp
```

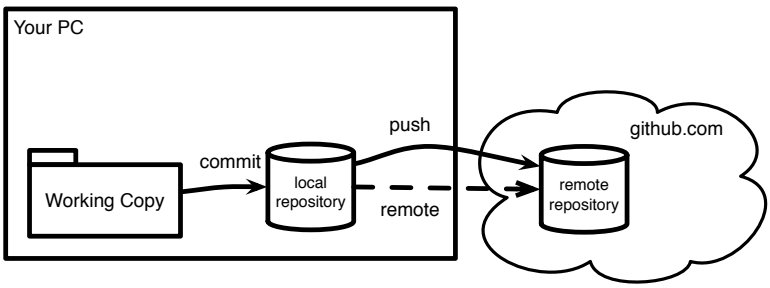

Figure 3-13 Push is an operation that sends commits from your local repository to a remote repository.

To avoid specifying the remote and destination branch on every push (which may be a bit verbose), you can avoid those parameters and rely on `git` default values. By default the `git push` operation will try to push to the **branch's upstream**. A branch's upstream is the per-branch configuration saying to which remote/branch pair it should push by default. When we clone a repository, the default branch comes with an already configured upstream. We can interrogate `git` for the branch's upstreams with the super verbose flag in the branch command, *i.e.*, `git branch -vv`, where we can see for example that our **master** branch's upstream is **origin/master**, while our **development** branch has no upstream.

```
$ git branch -vv   # doubly verbose!
  development   1656797 This commit adds a new feature
  master        f269295 [origin/master] First commit
```

On the other side, when a branch has no upstream, a push operation will by default fail with a `git` error.`git` will ask us to set an upstream, or otherwise

specify a pair remote/branch for each push.

```
$ git push
fatal: The current branch test has no upstream branch.
To push the current branch and set the remote as upstream, use

    git push --set-upstream origin test

$ git push --set-upstream origin test
Counting objects: 3, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 271 bytes | 0 bytes/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To git@github.com:[your_username]/[your_repo_name].git
   b6dcc3f..f269295  master -> test
```

Finally, another thing may happen while pushing: `git` may reject our changes.

```
$ git push
To git@github.com:guillep/test.git
 ! [rejected]        master -> master (fetch first)
error: failed to push some refs to
    'git@github.com:[your_username]/[your_repo_name].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.
```

As the error message says, the remote has changes that we do not have locally, so we need to update our repository first. This can be solved with a pull and a merge.

3.7 Exercises

1. **Exercise 1**. Get a repository with many commits and checkout the parent of the current commit. This will put you in "Detached HEAD" state. Solve it using a new branch.
2. **Exercise 2**. Try to merge your previous branch into your new branch. What kind of merge is it?
3. **Exercise 3**. Repeat the scenario of the first exercise, apply a change to your one of your files and commit it. Try to merge your previous branch into your new branch. What kind of merge is it?

4. **Exercise 4**. Create a new online repository and push your changes into it.
5. **Exercise 5**. What is the smaller set of steps you could imagine to create a conflict?

CHAPTER 4

Practical Git Scenarios

In this chapter we will show some `git` features that can help you in your daily use of `git`.

4.1 Before commit little helpers

You may add too fast a file to your staging area or index. Or you may want to discard the changes you did to your working copy to get back in the situation where no changes were made. The two problems can be handled as follows.

Remove Files from the Staging Area

It may happen that you went too fast and added a file to your staging area (remember the groceries' list) and that you changed your mind and do not want to add it anymore. Your file is not committed so a simple `reset` will do the job. You can either remove single files or everything.

```
$ git reset HEAD file
# Or everything
$ git reset HEAD .
```

Getting back your file as in your local repository

Another common scenario is that you modified a file of your working copy and your realize that it would be better to drop the changes and get back the file that is versioned in your local repository. Simply checkouting again will fix your problem

```
$ git checkout file
```

4.2 Exploring the History

The `git log` Command

The commit graphs we have shown so far are not evident at all while when we use the `git status` command. There is however a way to ask `git` about them using the `git log` command.

```
$ git log
commit 0c0e5ff55b56fe8eabc1661a1da64b41f9d74472
Author: Guille Polito <guillermopolito@gmail.com>
Date:   Wed Mar 21 15:37:32 2018 +0100

    Adding a title

commit 37adf4eaa945cbd7460991f88bff5aa902db06ce
Author: Guille Polito <guillermopolito@gmail.com>
Date:   Wed Mar 21 14:02:43 2018 +0100

    first version
```

`git log` prints the list of commits in order of parenthood. The one on the top is the most recent commit, our last commit. The one below is its parent, and so on. As you can see, each commit has an id, the author name, the timestamp and its message.

To display a more compact version (commit ids + message) of the log use

```
git log --oneline
```

We can also ask `git` what are the changes introduced in a particular commit using the command `git show`.

```
$ git show 0c0e5ff55b56fe8eabc1661a1da64b41f9d74472
commit 0c0e5ff55b56fe8eabc1661a1da64b41f9d74472
Author: Guille Polito <guillermopolito@gmail.com>
Date:   Wed Mar 21 15:37:32 2018 +0100

    Adding a title

diff ----git== a/README.md b/README.md
index e69de29..cad05f1 100644
--- a/README.md
+++ b/README.md
@@ -0,0 +1 @@
+! a title
\ No newline at end of file
```

That will give us the commit description as in `git log` plus a (not so readable) diff of the modified files showing the inserted, modified and deleted

lines. More advanced graphical tools are able to read this description and show a more user-friendly diff.

Accessing the History Graph

Git's log provides a more graphish view on the terminal using some cute ascii art. This view can be accesses through the `git log --graph --oneline --all` command. Here is an example of this view for a more complex project. In this view, stars represent the commits with their ids and commit messages, and lines represent the parenthood relationships.

```
$ git log --graph --oneline --all
* 4eb8446 Documenting
* e5a3e2e Add tests
* 680a79a Some other
| *   ed4854f Merge pull request #1137
| |\
| | * 9e30e37 Some feature
| * |   ba7f65c Merge pull request #1138
| |\ \
| | * | 31a40c4 Some Enhancement
| | |/
| * |   2d4698d Merge pull request #1139
| |\ \
| | * | 20c0ff4 Some fix
| | |/
| * |   ae3ec45 Merge pull request #1136
```

However, we are not always in the mood of using the terminal, or of wanting to decode what was done in ascii art. There are tools that are more suitable to explore the history of a project, usually providing some nice graphical capabilities. This is the case of tools such as SourceTree (Figure 4-1) or Github's network view (Figure 4-2).

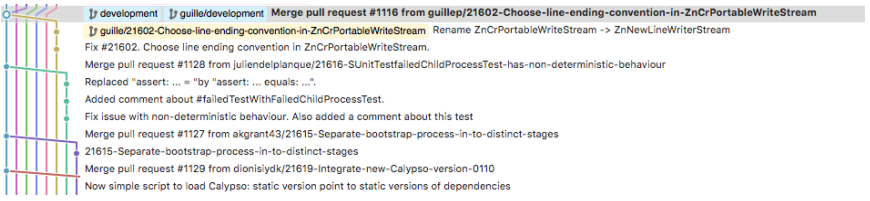

Figure 4-1 Example of SourceTree's commit graph view.

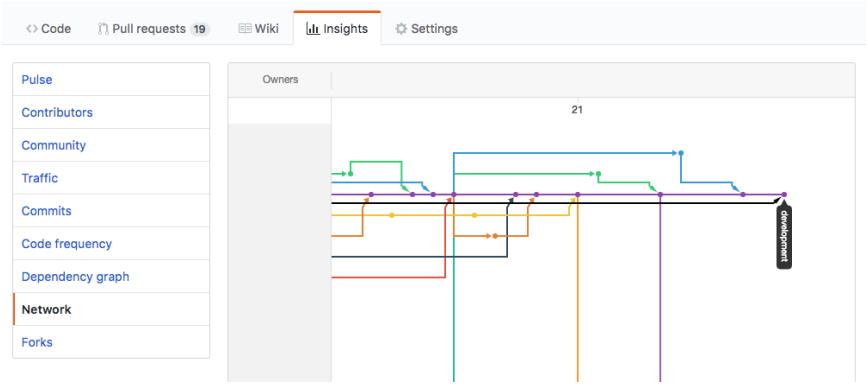

Figure 4-2 Example of Github's commit graph view.

4.3 Discarding your Local Committed Changes

It comes the time for every woman/man to make mistakes and want to discard them. Doing so may be dangerous, since once discarded you will not able to recover your changes. It is however possible to instruct `git` to do so. For it, there are two `git` comments that will perform the task for you and when combined they will completely discard every dirty file and directory in your repository:`git reset` and `git clean`. `commit_id` is the commit where you want to be.

```
$ git reset --hard <commit_id>
$ git clean -df
```

The `-d` option removes untracked directories in addition to untracked files, while the `-f` option is a shortcut `--force`, forcing the corresponding deletions.

If you want to just revert the last commit you can also use

```
$ git reset --hard HEAD~1
$ git clean -df
```

The reason for needing two commands instead of one relies on the fact that `git` has several staging areas (such as the ones used to keep the tracked files), which we usually would like to clean when we discard the repository. Of course, experienced readers may search why they would need both in Git's documentation.

4.4 Ignoring Files

Many times we will find that we do not want to commit some files that are in our repository's directory. This is mostly the case of generated or automatically downloaded files. For example, imagine you have a C project and some makefiles to compile it, generating a binary library. While it would be good to store the result of compilation from time to time, storing it in a `git` repository (or SVN, or Bazar) may be a cause of headaches. First, as you will see in 3, this may be a cause for conflicts. Second, since we should be able to generated such binary library from the sources, having the already compiled result in the repository does not add so much value.

This same ideas can be used to ignore any kind of generated file. For example, pdfs generated by document generation tools, meta-data files generated by IDEs and tools (e.g., Eclipse), compiled libraries (e.g., dll, so, or dylib files).

In such cases, we can tell `git` to ignore cetain files using the `.gitignore` file. The `.gitignore` file is an optional text file that we can write in the root of our repository with a list of file paths to ignore.

```
# Example of .gitignore file

# Lines starting with hashtags are comments

# A file name will ignore that file
someignoredfile.txt

# A file name will ignore that file
someignoredfile.txt

# A file pattern will ignore all pdf files
*.pdf
```

Once your file is ready, you have to add it and commit it to `git` to make it take effect.

```
$ git add .gitignore
$ git commit -m "Added gitignore"
```

From this moment on, all listed files will be ignored by `git add` and `git status`. And you will be able to perform further commands to add "all but ignored files":

```
$ git add .
```

If a file or a file type is tracked but you want `git` to ignore its changes afterward, adding it to .gitignore file will not make the job i.e. `git` will continue to track it. To avoid keeping track of it in the future, but secure it locally in your working directory, it must be removed from the tracking list using ==git rm –cached <file> (.<file_type>)==. Nevertheless, be aware that the file is still present in the past history!

4.5 Commiting a File Filtered out by the .gitignore

For certain workflow, it is better to filter out certain files such as generated pdf but from time to time you may want to version a file that would not be because its extension or folder is listed in the `.gitignore` file. You can still commit such a such without having to touch the `.gitignore` file. You can use the `--force` option of the `add` command.

```
$ git add -f that.pdf
$ git commit -m "that file is still important"
```

4.6 Getting out of Detached HEAD

Detached head means no other than "HEAD is not pointing to a branch". Being in a detached HEAD state is not bad in itself, but it may provoke loss of changes. As a matter of fact, any commit that is not properly referenced by another commit or by another `git` reference (tag, branch) may be garbage collected.

`git` will not forbid you to commit in this state, but any new commit you create will only reachable if you remember the commit hash. To get out of dettached HEAD, the easiest solution is to checkout a branch, as we will see in the next section. Checking out a branch will set HEAD to point to a branch instead of a commit, saving you some HEADaches.

4.7 Accessing your Repository through SSH

To be able to access your repository from your local machine, you need to setup your credentials. Think it this way: you need to tell the server who you are on every interaction you have with it. Otherwise, Github will reject any operation against your repository. Such a setup requires the creation and uploading of SSH keys.

An SSH key works as a lock: a key is actually a pair of a public and a private key. The private key is meant to reside in your machine and not be published at all. A public key is meant to be shared with others to prove your identity. Whenever you want to prove your identity, SSH will exchange messages encrypted with your public key, and see if you are able to decrypt it using your private key.

To create an SSH key, in *nix systems you can simply type in your terminal

```
$ ssh-keygen -t rsa -b 4096 -C "your_email@some_domain.com"
```

Follow the instructions in your terminal such as setting the location for your key pair (usually it is `$HOME/.ssh`) and the passphrase (a kind of password). Finally, you'll end up with your public/private pair on the selected location. It is now time to upload it to Github.

Connect yourself to your Github settings and go to the "SSH and GPG keys" menu. Import there the contents of your public key file. You should be now able to use your repository.

4.8 Rewriting the History

Many times it happens that we accidentally commit something wrong. Maybe we wanted to commit more or less things, maybe a completely different content, or we did a mistake in the commit's message. In these cases, we can rewrite Git's history, e.g, undo our current commit and go back to the previous commit, or rewrite the current commit with some new properties.

Be careful! Rewriting the history can have severe consequences. Imagine that the commit you want to undo was already pushed. This means that somebody else could have pulled this commit into her/his repository. If we undo this already published commit, we are making everybody else's repositories obsolete! This can be indeed problematic depending on the number of users the project has, and their knowledge on `git` to be able to solve this issue.

Undo a commit using `git reset --hard`

To undo the last commit, it is as easy as:

```
$ git reset --hard HEAD~1
```

`git reset --hard [commitish]` makes your current branch point to [commitish]. `HEAD` is your current head, and you can read `~1` as "minus one". In other words, `HEAD~1` is head minus one, which boils down to the parent of head, our previous commit.

You can use this same trick to rewrite the history in any other way, since you can use any commitish expression to reset. For example, `HEAD~17` means 17 versions before head, or `someBranch~4` means four commits before the branch `someBranch`.

Update a Commit's Message using `git commit --amend`

To change our current commit's message you can use the following command:

```
$ git  commit --amend -m "New commit message"
```

Or, if you don't use the `-m` option, a text editor will be prompt so you can edit a commit message.

```
$ git  commit --amend
```

You can use the same trick not only to modify a commit's message but to modify your entire commit. Actually, just adding new things with `git add`

before an `--amend` will replace the current commit with a new commit merging the previous commit changes with what you just added.

4.9 How to Overwrite/Modify Commits

WARNING: It is highly not recommended to rewrite the history of a repo especially when part of it has already been pushed to a remote. Modifying the history will most likely break the history shared by the different collaborators and you may deal with an inextricable merge conflict.

Change the last Commit

Imagine you have just committed your changes and have not pushed them yet, but

1. you are not satified with the commit message

```
$ git  log --oneline
$ git  commit --amend -m "Updated commit message"
$ git  log --oneline
```

1. you forgot to save some modification or to add some files before committing. Then make your changes and use

```
$ git  commit -a --amend --no-edit
```

Merge two commits

First, it is worth repeating that you must think twice before modifying the history of the repo. Now, assume that you have not pushed the corresponding commits. Merging two consecutive commits is a way to work with a cleaner tree. Consider you want to merge commits ”Intermediate” and ”Old”

```
$ git  log --oneline

eae7846 New
71c0c64 Intermediate
f039832 Old
cca92f1 Even older
```

Then, you can interactively `-i` focus on the last three `HEAD~3` commit.

```
$ git  rebase -i HEAD~3

pick f039832 Old
pick 71c0c64 Intermidiate
pick eae7846 New
```

Observe that the commits are displayed in the reversed order. Now you can squash the commit "Intermediate" into its parent commit "Old"

```
pick f039832 Old
squash 71c0c64 Intermidiate
pick eae7846 New
```

And set the message e.g. "Merge intermediate + old" attached to the single.

```
# This is a combination of 2 commits.
# This is the 1st commit message:

Old

# This is the commit message #2:

Intermediate

# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
```

```
$ git  log --oneline

eae7846 New
651375a Merge intermediate + old
cca92f1 Even older
```

Note that the commit id has changed

Pushing Rewritten History

As soon as the history we have rewritten was never pushed before, we can continue working normally and pushing our changes then without problems. However, if we have already pushed the commit we want to undo, this means that we are potentially impacting all users of our repository. Because of the problems it can pose to other people, pushing a rewritten history is not a completely favoured by `git`. Better said, it is not allowed by default and you'll be warned about it:

```
$ git  push
To git@github.com:REPOSITORY_OWNER/YOUR_REPOSITORY.git
 ! [rejected]        YOUR_BRANCH -> YOUR_BRANCH (non-fast-forward)
error: failed to push some refs to
    'git@github.com:REPOSITORY_OWNER/YOUR_REPOSITORY.git'
hint: Updates were rejected because the tip of your current branch
    is behind
hint: its remote counterpart. Integrate the remote changes (e.g.
hint: '==git== pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in '==git== push --help'
    for details.
```

With this message `git` means that you should not blindly overwrite the history. Also, it suggests to pull changes from the remote repository. However, doing that will bring back to our repository the history we wanted to undo! What we want to do is to impose our current (undone) state in the remote repository. To do that, we need to **force** the push using the `git push --force` or the `git push -f` option.

```
$==git== push -f
Total 0 (delta 0), reused 0 (delta 0)
To git@github.com:REPOSITORY_OWNER/YOUR_REPOSITORY.git
+ a1713f3...6e0c7bf YOUR_BRANCH -> YOUR_BRANCH (forced update)
```

4.10 Conclusion

This chapter presented some operations that may help you. As a general principle avoid rewriting history because you may lose changes and get into unrecovable state.

CHAPTER 5

Publishing your first Pharo project with Iceberg

In this chapter, we explain how you can publish your project on GitHub using Iceberg. We do not explain basic concepts like commit, push/pull, merging, or cloning.

A strong precondition before reading this chapter is that you must be able to publish from the command line to the `git` hosting service that you want to use. If you cannot do not expect Iceberg to fix it magically for you. Let us get started.

5.1 For the impatient

If you do not want to read everything, here is the executive summary.

- Create a project on GitHub or any git-based platform.
- [Optional] Configure Iceberg to use custom SSH keys.
- Add a project in Iceberg.
  - Optionally but strongly recommended, in the cloned repository, create a directory named `src` on your file system. This is a good convention.
- In Iceberg, open your project and add your packages.
- Commit to your project.
- [Optional] Add a baseline to ease loading your project.
- Push your change to your remote repository.

You are done. Now we can explain calmly.

5.2 Basic Architecture

As `git` is a distributed versioning system, you need a *local* clone of the repository and a *working copy*. Your working copy and local repository are usually on your machine. This is to this local repository that your changes will be committed to before being pushed to remote repositories (Figure 5-1). We will see in the next Chapter that the situation is a bit more complex and that Iceberg is hiding the extra complexity for us.

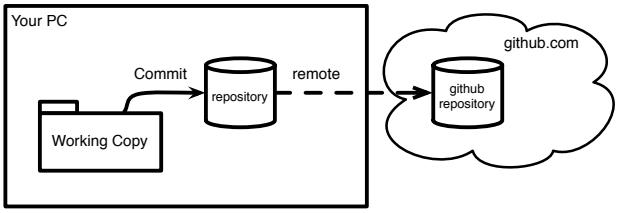

Figure 5-1 A distributed versioning system.

5.3 Create a new project on GitHub

While you can save locally first and then later create a remote repository, in this chapter we first create a new project on GitHub. Figure 5-2 shows the creation of a project on GitHub. The order does not matter. What is different is that you should use different options when addin a repository to Iceberg as we will show later.

5.4 [Optional] SSH setup: Tell Iceberg to use your keys

To be able to commit to your `git` project, you should either use HTTPS or you will need to set up valid credentials in your system. In case you use SSH (the default way), you will need to make sure those keys are available to your GitHub account and also that the shell adds them for smoother communication with the server. See the Chapter 9 Tips and Tricks for some help with setting up your ssh keys.

Go to settings browser, search for "Use custom SSH keys" and enter your data there as shown in Figure 5-3).

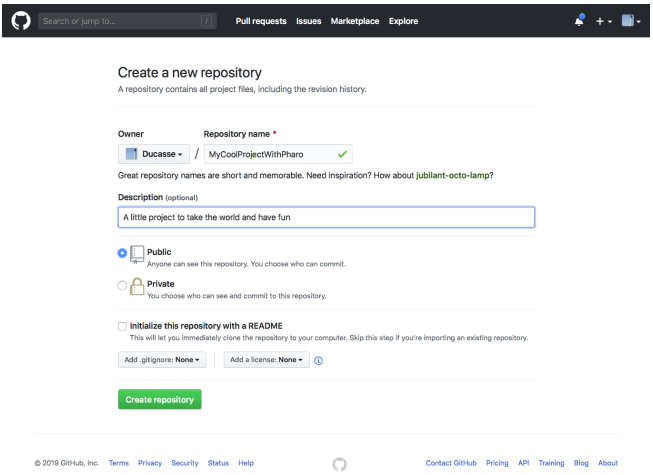

Figure 5-2 Create a new project on GitHub.

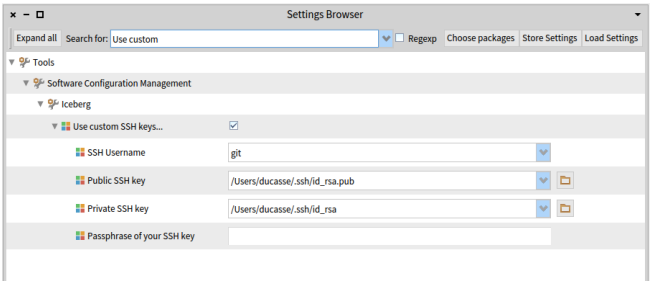

Figure 5-3 Use Custom SSH keys settings.

Alternatively, you can execute the following expressions in your image playground or add them to your Pharo system preference file (See Menu System item startup):

```
IceCredentialsProvider useCustomSsh: true.
IceCredentialsProvider sshCredentials
  publicKey: 'path\to\ssh\id_rsa.pub';
  privateKey: 'path\to\ssh\id_rsa'
```

Pro Tip: This can be used too in case you have a non-default key file. You just need to replace `id_rsa` with your file name.

5.5 Iceberg *Repositories* browser

Figure 5-4 shows the top-level Iceberg pane. It shows that for now you do not have defined nor loaded any project. It shows the Pharo project and indicates that it could not find its local repository by displaying 'Local repository missing'.

First, you do not have to worry about the Pharo project or repository if you do not want to contribute to Pharo. So just go ahead. Now if you want to understand what is happening here is the explanation. The Pharo system does not have any idea where it should look for the `git` repository corresponding to the source of the classes it contains. Indeed, the image you are executing may have been built somewhere, patched or not many times. Now Pharo is fully operational without having a local repository. You can browse system classes and methods because Pharo has its own internal source management. This warning indicates that if you want to version Pharo system code using `git` then you should indicate to the system where the clone and working copy are located on your local machine. So if you do not plan to modify and version the Pharo system code, you do not have to worry.

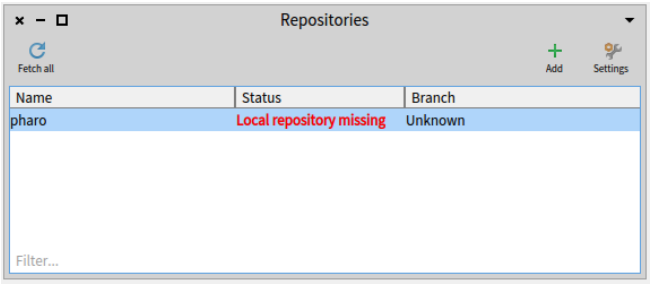

Figure 5-4 Iceberg *Repositories* browser on a fresh image indicates that if you want to version modifications to Pharo itself you will have to tell Iceberg where the Pharo clone is located. But you do not care.

5.6 Add a new project to Iceberg

The first step is then to add a project to Iceberg:

- Press the '+' button to the right of the Iceberg main window.
- Select the source of your project. In our example, since you did not clone your project yet, choose the GitHub option.

Notice that you can either use SSH (Figure 5-5) or HTTPS (Figure 5-6).

Figure 5-5 and 5-6) instruct Iceberg to clone the repository we just created on GitHub. We specify the owner, project, and physical location where the local clone and `git` working copy will be on your disk.

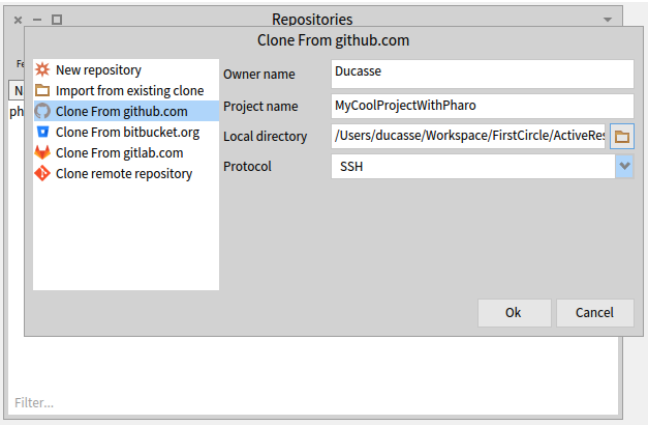

Figure 5-5 Cloning a project hosted on GitHub via SSH.

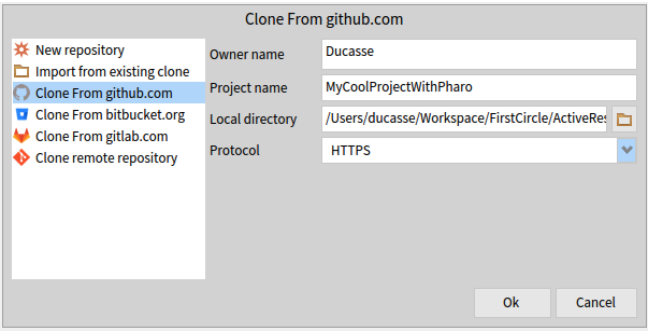

Figure 5-6 Cloning a project hosted on GitHub via HTTPS.

Iceberg has now added your project to its list of managed projects and cloned an empty repository to your disk. You will see the status of your project, as in Figure 5-7. Here is a breakdown of what you are seeing:

- MyCoolProjectWithPharo has a star and is green. This usually means that you have changes which haven't been committed yet, but may also happen in unrelated edge cases like this one. Don't worry about this for now.
- The Status of the project is 'No Project Found' and this is more important. This is normal since the project is empty. Iceberg cannot find its

metadata. We will fix this soon.

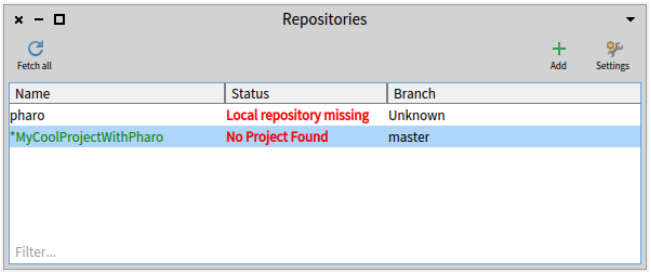

Figure 5-7 Just after cloning an empty project, Iceberg reports that the project is missing information.

Later on, when you will have committed changes to your project and you want to load it in another image, when you clone again, you will see that Iceberg reports that the project is not loaded as shown in Figure 5-8.

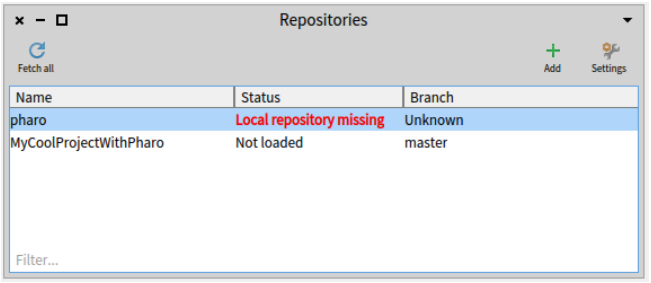

Figure 5-8 Adding a project with some contents shows that the project is not loaded - not that it is not found.

5.7 Repair to the rescue

Iceberg is a smart tool that tries to help you fix the problems you may encounter while working with `git`. As a general principle, each time you get a status with red text (such as ”No Project Found” or ”Detached Working Copy”), you should ask Iceberg to fix it using the **Repair** command.

Iceberg cannot solve all situations automatically, but it will propose and explain possible repair actions. The actions are ranked from most to least likely to be right one. Each action has a displayed explanation of the situation and the consequences of using it. It is always a good idea to read them. Setting your repository the right way makes it extremely hard to lose any piece of

code with Iceberg and Pharo is general since Pharo contains its own copy of the code.

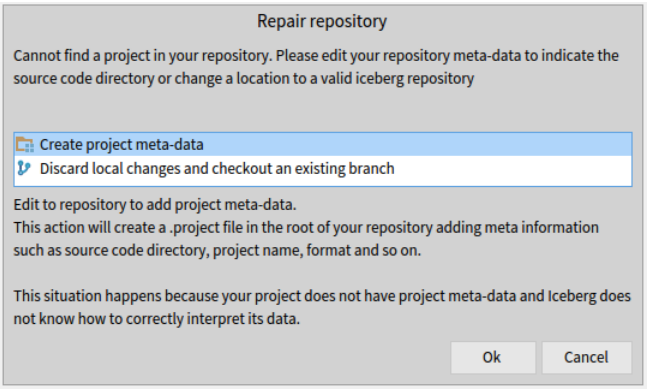

Figure 5-9 Create project metadata action and explanation.

5.8 Create project metadata

Iceberg reported that it could not find the project because some metadata were missing such as the format of the code encodings and the example location inside the repository. When we activate the repair command we get Figure 5-9. It shows the "Create project metadata" action and its explanation.

When you choose to create the project metadata, Iceberg shows you the filesystem of your project as well as the repository format as shown in Figure 5-10. Tonel is the preferred format for Pharo projects. It has been designed to be Windows and file system friendly. Change it only if you know what you are doing!

Before accepting the changes, it is a good idea to add a source (`src`) folder to your repository. Do that by pressing the + icon. You will be prompted to specify the folder for code as shown in Figure 5-11. Iceberg will show you the exact structure of your project as shown in Figure 5-12.

After accepting the project details, Iceberg shows you the files that you will be committing as shown in Figure 5-13

Once you have committed the metadata, Iceberg shows you that your project has been repaired but is not loaded as shown in Figure 5-8. This is normal since we haven't added any packages to our project yet. You can optionally push your changes to your remote repository.

Your local repository is ready, let's move on to the next part.

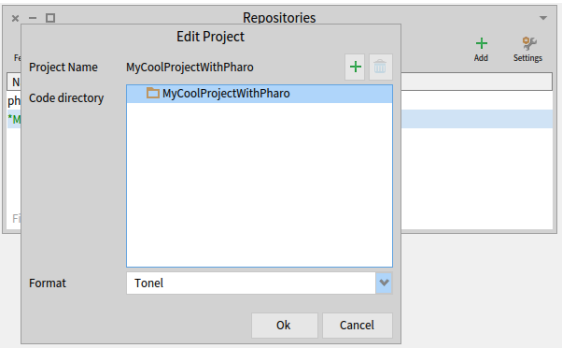

Figure 5-10 Showing where the metadata will be saved and the format encodings.

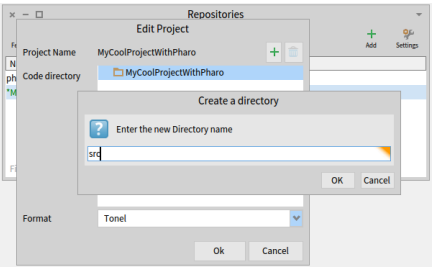

Figure 5-11 Adding a src repository for code storage.

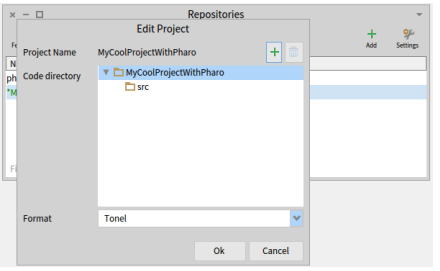

Figure 5-12 Resulting situation with a src folder.

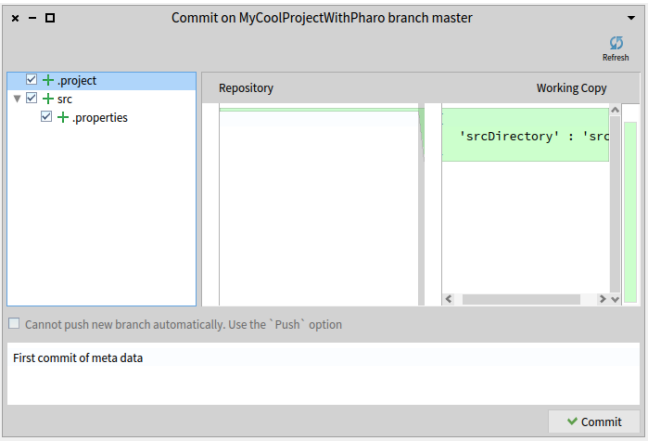

Figure 5-13 Details of metadata commit.

5.9 Add and commit your package using the *Working copy* browser

Once your project contains Iceberg metadata, Iceberg will be able to manage it easily. Double click on your project to bring a *Working copy* browser for your project. It lists all the packages that compose your project. Right now you have none. Add a package by pressing the + (Add Package) iconic button as shown by Figure 5-14.

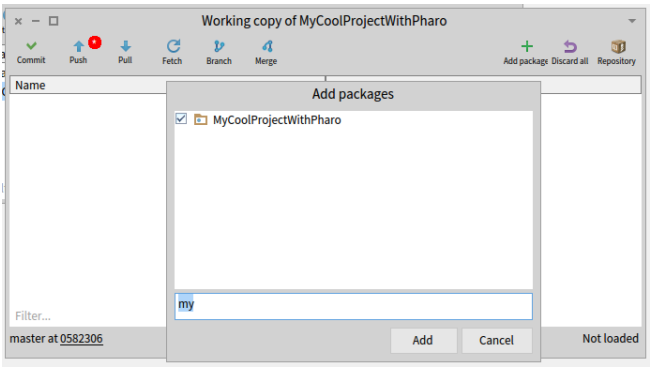

Figure 5-14 Adding a package to your project using the *Working copy* browser.

Again, Iceberg shows that your package contains changes that are not committed using the green color and the star in front of the package name as shown in Figure 5-15.

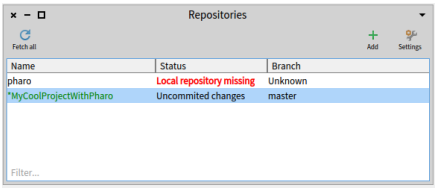

Figure 5-15 Iceberg indicates that your package has unsaved changes – indeed you just added your package.

Commit the changes

Commit the changes to your local repository using the Commit button as shown in Figure 5-16. Iceberg lets you choose the changed entities you want to commit. Here this is not needed but this is an important feature. Iceberg will show the result of the commit action by removing the star and changing the color. It now shows that the code in the image is in sync with your local repository as shown by Figure 5-17. You can commit several times if needed.

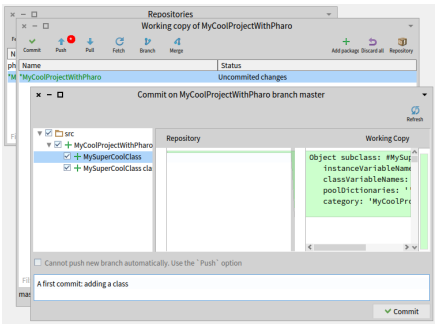

Figure 5-16 When you commit changes, Iceberg shows you the code about to be committed and you can chose the code entities that will effectively be saved.

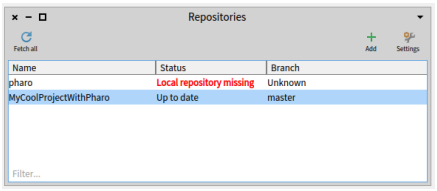

Figure 5-17 Once changes committed, Iceberg reflects that your project is in sync with the code in your local repository.

Publish your changes to your remote

Now you are nearly done. Publish your changes from your local directory to your remote repository using the Push button. You may be prompted for credentials if you used HTTPS.

When you push your changes, Iceberg will show you all the commits awaiting publication and will push them to your remote repository as shown in Figure 5-18. The figure shows the commits we are about to make to add a baseline, which will allow you to easily load your project in other images.

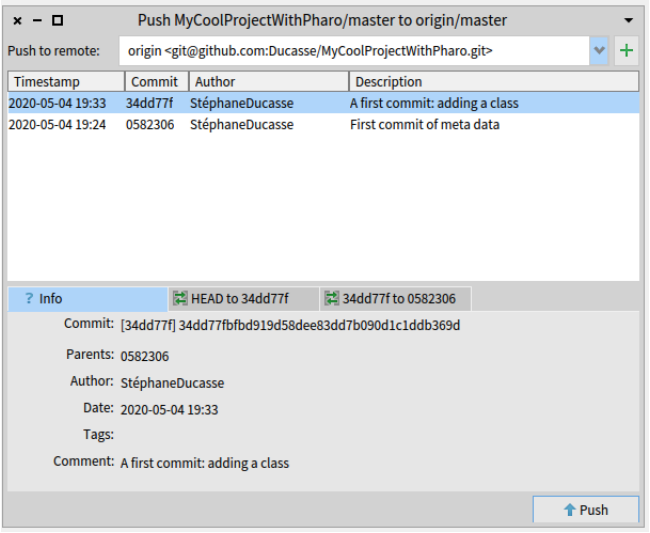

Figure 5-18 Publishing your committed changes.

Now you are basically done.

5.10 Conclusion

You now know the essential aspects of managing your code with GitHub. Iceberg has been designed to guide you so please listen to it unless you really know what you are doing. You are now ready to use services offered around GitHub to improve your code control and quality!

CHAPTER 6

Configure your project nicely

Versioning code is just the first part of making sure that you and others can reload your code. In this chapter we describe how to define a baseline, a project map that you will use to define dependencies within your project and dependencies to other projects. We also show how to add a good `.gitignore` file. In the next chapter, we will show how to configure your project to get more out of the services offered within the GitHub ecosystem such as Travis-ci to execute automatically your tests.

We start by showing you how you can commit your code if you did not create your remote repository first.

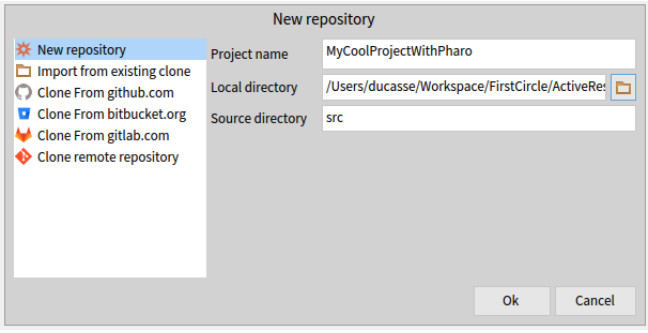

Figure 6-1 Creating a local repository without pre-existing remote repository.

6.1 What if I did not create a remote repository

In the previous chapter, we started by creating a remote repository on GitHub. Then we asked Iceberg to add a project by cloning it from GitHub. Now you may ask yourself about the process to publish first your project locally without a pre-existing repository. This is actually simple.

Create a new repository.

When you add a new repository use the 'New repository' option as shown in 6-1.

Add a remote.

If you want to commit to a remote repository, you will have to add it using the *Repository* browser. You can access this browser through the associated menu item or the icon. The *Repository* browser gives you access to the `git` repositories associated with your project: you can access, manage branches and also add or remove remote repositories. Figure 6-3 shows the repository browser on our project.

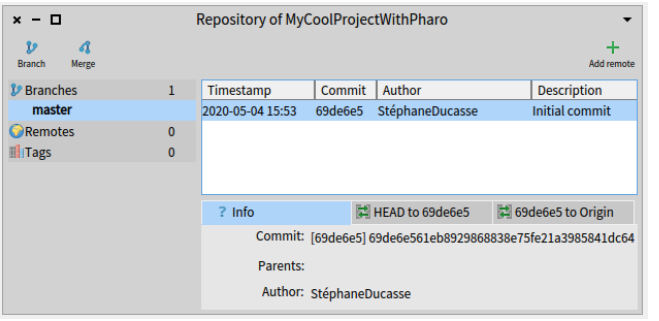

Figure 6-2 Opening the repository browser let you add and browse branches as well as remote repositories.

Pressing on the 'Add remote' iconic button adds a remote by filling the needed information that you can find in your GitHub project. Figure 6-3 shows it for the sample project using SSH and Figure 6-4 for HTTPS.

Push to the remote.

Now you can push your changes and versions to the remote repository using the Push iconic button. Once you have pushed you can see that you have one remote as shown in Figure 6-5.

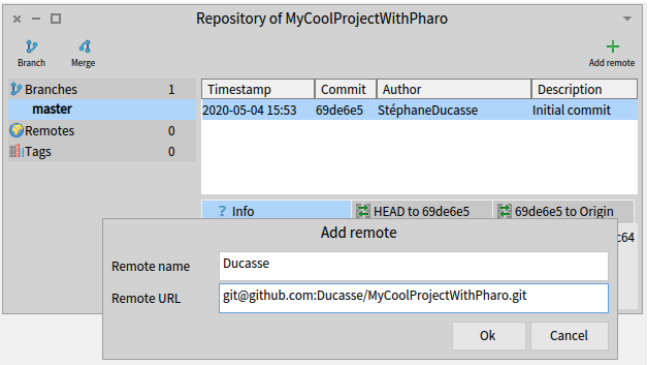

Figure 6-3 Adding a remote using the *Repository* browser of your project (SSH version).

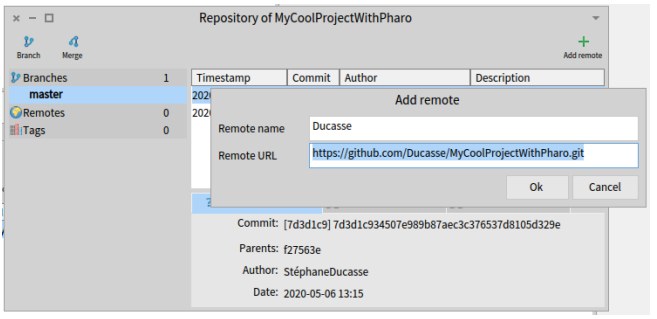

Figure 6-4 Adding a remote using the *Repository* browser of your project (HTTP version).

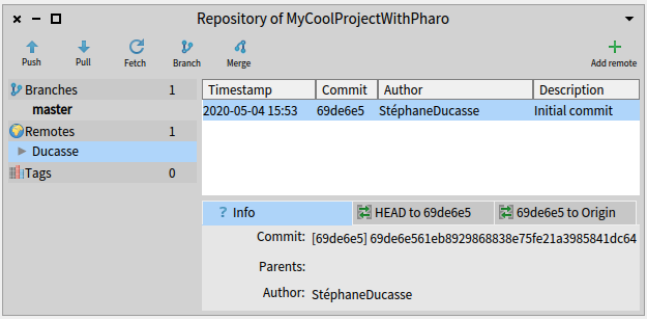

Figure 6-5 Once you pushed you changes to the remote repository.

6.2 Configuring automatically Pharo to commit in HTTP-S/SSH

With recent version of github, you should use a token to connect using HTTPS to be able to commit to your repository. Now you can configure Pharo to store your token as well your github account authentification using startup preferences.

In addition you can configure Pharo to point to your private and public SSH key as follows:

```
StartupPreferencesLoader default executeAtomicItems: {
    StartupAction
        name: 'Logo'
        code: [ PolymorphSystemSettings showDesktopLogo: false] .
    StartupAction
        name: 'Git Settings'
        code: [
            Iceberg enableMetacelloIntegration: true.
            IceCredentialStore current
                    storeCredential: (IcePlaintextCredentials new
                    username: 'xxJohnDoe';
                    password: 'xxJohnDoePassword';
                    host: 'github.com';
                    yourself).
            IceCredentialsProvider sshCredentials
                username: 'git';
                publicKey: 'Path to your public rsa.pub file (public
    key)';
                privateKey: 'Path to your private rsa file (private
    key)'.
            IceCredentialStore current
                storeCredential: (IceTokenCredentials new
                    username: 'xxJohnDoe';
                    token: 'magictoken here ';
                    yourself)
                forHostname: 'github.com'.
            ].
}
```

You should place this file in the preference folder that depends on your OS. You can find this place by using the `Startup` menu. Note that you can configure this for all or one specific version of Pharo.

6.3 Defining a `BaselineOf`

A *baseline* is a description of the architecture of a project. You will express the dependencies between your packages and other projects so that all the

dependent projects are loaded without the user having to understand them or the links between them.

A baseline is expressed as a subclass of `BaselineOf` and packaged in a package named `'BaselineOfXXX'` (where 'XXX' is the name of your project). So if you have no dependencies, you can have something like this.

```
BaselineOf subclass: #BaselineOfMyCoolProjectWithPharo
  ...
  package: 'BaselineOfMyCoolProjectWithPharo'
```

```
BaselineOfMyCoolProjectWithPharo >> baseline: spec
  <baseline>
  spec
    for: #common
    do: [ spec package: 'MyCoolProjectWithPharo' ]
```

Once you have defined your baseline, you should add its package to your project using the working copy browser as explained in the previous chapter. You should obtain the following situation shown in Figure 6-6. Now, commit it and push your changes to your remote repository.

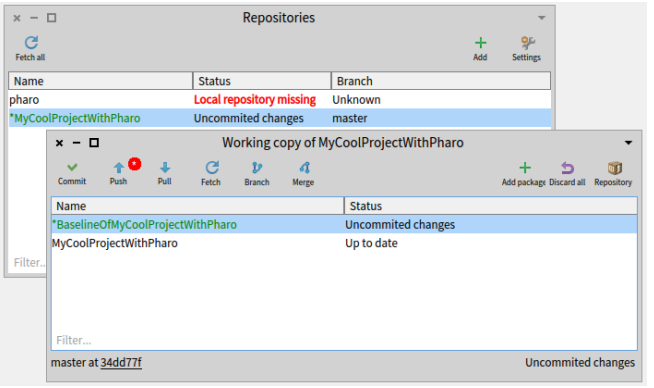

Figure 6-6 Added the baseline package to your project using the *Working copy* browser.

A more elaborated web resource about baseline possibility is available at: https://github.com/pharo-open-documentation/pharo-wiki/.

6.4 Loading from an existing repository

Once you have a repository you committed code to and would like to load it into a new Pharo image, there are two ways to work this out.

Manual load.

- Add the project as explained in the first chapter
- Open the working copy browser by double-clicking on the project line in the repositories browser.
- Select a package and manually load it.

Scripting the load.

The second way is to make use of Metacello. However, this will only work if you have already created a `BaselineOf`. In this case, you can just execute the following snippet:

```
Metacello new
  baseline: 'MyCoolProjectWithPharo';
  repository: 'github://Ducasse/MyCoolProjectWithPharo/src';
  load
```

For projects with metadata, like the one we just created, that's it. Notice that we not only mention the GitHub pass but also added the code folder (here `src`).

6.5 [Optional] Add a nice .gitignore file

Iceberg automatically manages files such as .gitignore.

```
# For Pharo 70 and up
# http://www.pharo.org
# Since Pharo 70 all the community is moving to git.

# image, changes and sources
*.changes
*.sources
*.image

# Pharo Debug log file and launcher metadata
PharoDebug.log
pharo.version
meta-inf.ston

# Since Pharo 70, all local cache files for Monticello package
    cache, playground, epicea... are under the pharo-local
/pharo-local

# Metacello-github cache
/github-cache
github-*.zip
```

6.6 Going further: Understanding the architecture

As `git` is a distributed versioning system, you need a local clone of your repository. In general, you edit your working copy located on your hard drive and you commit to your local clone, and from there you push to remote repositories like GitHub. We explain here the specificity of managing Pharo with git.

When coding in Pharo, you should understand that you are not directly editing your local working copy, you are modifying objects that represent classes and methods that are living in the Pharo environment. Therefore it is like you have a double working copy: Pharo itself and the `git` working copy.

When you use `git` command lines, you have to understand that there is the code in the image and the code in the working copy (and your local clone).To update your image, you *first* have to update your `git` working copy and *then* load code from the working copy to the image. To save your code you have to save the code to files, add them to your git working copy and commit them to your clone.

Now the interesting part is that Iceberg manages all this for you transparently. All the synchronization between these two working copies is done behind the scene.

Figure 6-7 shows the architecture of the system.

- You have your code in the Pharo image.
- Pharo is acting as a working copy (it contains the contents of the local `git` repository).
- Iceberg manages the publication of your code to the `git` working copy and the `git` local repository.
- Iceberg manages the publication of your code to remote repositories.
- Iceberg manages the re-synchronization of your image with the `git` local repository, `git` remote repositories and the `git` working copy.

6.7 Conclusion

We show how to package your code correctly. It will help you to reload it and use the services that we will present in the next chapter.

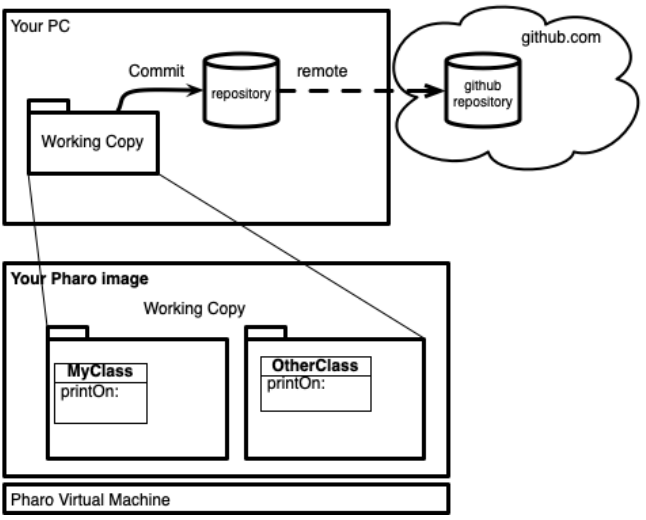

Figure 6-7 Architecture.

CHAPTER 7

Empowering your projects

Now that you can save your code on GitHub in a breeze, you can take advantage of services to automate actions. For example, to execute tests using Travis) or AppVeyor for Windows.

7.1 Adding Travis integration

By adding two simple files, you can have the tests of your project automatically run after each commit with Travis. You need to enable Travis in your GitHub repository. Since Travis is changing its policy and work tightly in relation with GitHub you might have a better time checking on their respective websites.

You should also add the two following files: `.travis.yml` and `.smalltalk.ston` in the top level of your repository.

`.travis.yml`

```
language: smalltalk
sudo: false
os:
  - linux
smalltalk:
  - Pharo-7.0
```

`.smalltalk.ston`

```
SmalltalkCISpec {
  #loading : [
    SCIMetacelloLoadSpec {
      #baseline : 'MyCoolProjectWithPharo',
      #directory : src',
```

```
      #platforms : [ #pharo ]
    }
  ]
}
```

If you have done everything right, Travis will pick up the changes and will start testing and building your project... and you are done, congratulations!

7.2 On windows

If you want to make sure that your code runs on Windows, you should use the Appveyor service and add the `appveyor.yml` file.

```
environment:
  CYG_ROOT: C:\cygwin
  CYG_BASH: C:\cygwin\bin\bash
  CYG_CACHE: C:\cygwin\var\cache\setup
  CYG_EXE: C:\cygwin\setup-x86.exe
  CYG_MIRROR: http://cygwin.mirror.constant.com
  SCI_RUN: /cygdrive/c/smalltalkCI-master/run.sh
  matrix:
    - SMALLTALK: Pharo-6.1
    - SMALLTALK: Pharo-7.0

platform:
  - x86

install:
  - '%CYG_EXE% -dgnqNO -R "%CYG_ROOT%" -s "%CYG_MIRROR%" -l
    "%CYG_CACHE%" -P unzip'
  - ps: Start-FileDownload
    "https://GitHub.com/hpi-swa/smalltalkCI/archive/master.zip"
    "C:\smalltalkCI.zip"
  - 7z x C:\smalltalkCI.zip -oC:\ -y > NULL

build: false

test_script:
  - '%CYG_BASH% -lc "cd $APPVEYOR_BUILD_FOLDER; exec 0</dev/null;
    $SCI_RUN"'
```

7.3 Adding badges

With CI happily running, you can add a badge to your `README` that will show the current status of your project. Here is the badge of the Containers-Stack project where we also enabled the `coveralls.io` test coverage service.

```
# Containers-Stack
A dead stupid stack implementation, but one fully working :)

[![Build
    Status](https://travis-ci.com/Ducasse/Containers-Stack.svg?branch=master)]
(https://travis-ci.com/Ducasse/Containers-Stack)
[![Coverage
    Status](https://coveralls.io/repos/GitHub//Ducasse/Containers-Stack/badge.svg
(https://coveralls.io/GitHub//Ducasse/Containers-Stack?branch=master)
[![License](https://img.shields.io/badge/license-MIT-blue.svg)]()
[![Pharo
    version](https://img.shields.io/badge/Pharo-7.0-%23aac9ff.svg)]
(https://pharo.org/download)
[![Pharo
    version](https://img.shields.io/badge/Pharo-8.0-%23aac9ff.svg)]
(https://pharo.org/download)

## Installation
The following script installs Containers-Stack in Pharo.
```

Metacello new baseline: 'ContainersStack'; repository: 'GitHub://Ducasse/Containers-Stack/src'; load.

To obtain the necessary link, click on the badge in your Travis project overview and select one of the options. You can insert the markdown code directly into your README.md.

7.4 Conclusion

With a continuous integration setup, you make sure that you can run your software on different platforms. In addition, you make sure that you will be able to smoothly load your code and that any additions or modifications you just did is not disrupting the whole project. Other services such as coveralls are available for Pharo.

CHAPTER 8

Contributing to Pharo

Pharo itself is hosted on GitHub under http://www.github.com/pharo-project/. All the development of the core system is done via this code repository. You can contribute to Pharo itself by fixing some issues. The official and unique way of submitting code is by doing a *Pull Request* from your fork to the official repository.

You can see the issues on the Pharo Github repository: http://www.github.com/pharo-project/pharo/issues.

8.1 In a nutshell

The general process is summarised by the following steps:

1. Download the latest version (this is important since I will make sure that you do not have to resync your fork).
2. Use repair (it will fetch some commits and make sure that your local repository is in sync with the remotes).
3. Use repair (and create a branch – let us called it Bottom-123 – that will point to the commits of your image).
4. Create a new branch for the issue you want to work on.
5. Code, then commit, then push to YOUR fork
6. Issue a PullRequest from your fork to Pharo
7. Check out your Bottom-123 branch and you can restart step 5

8.2 Step 0: Setting up the development environment

Download and launch the latest Pharo version:

The recommended way to download Pharo is to use the Pharo launcher, which you can get from Pharo download's page. Pro tip: If you're on Windows and you want a nice command line environment, install msys else you can try the following patterned expression:

```
curl https://get.pharo.org/[32/64]/[version]+vm | bash
./pharo-ui Pharo.image
```

If you don't have a command line environment, you can download both image and vm manually from the following links for example: http://files.pharo.org/get-files/120/

Browse the file server and find the files that suit your environment.

8.3 Fork the Pharo repository

All changes you'll make will be versioned in *your own* fork of the Pharo repository. Then, from your fork you'll be able to issue pull requests to Pharo, where they will be reviewed, and luckily, integrated.

Go to Pharo GitHub's repository and click on the fork button on the top right. Yes, this means that you'll need a github account to contribute to Pharo, yes.

8.4 Setup Iceberg

To be able to contribute to Pharo, you need a Pharo git repository.

You will need to set a valid set of credentials in your system to be able to work with Pharo. In case you use SSH (the default way), you will need to make sure those keys are available but you can start by using HTTPS since this is easier. In case they are not (and you will notice as soon as you try to clone a project or commit a change into one), you can add them following these steps:

Linux setting credentials

Execute in your shell:

```
    ssh-add ~/.ssh/id_rsa
```

OSX setting credentials

Execute in your shell:

```
    ssh-add -K ~/.ssh/id_rsa
```

Both for OSX and Linux you can add such lines in your `.bash_profile` (or the one corresponding to your shell installation) so they are automatically executed on each new shell session.

Windows setting credentials

Windows is more complicated, you may need to generate a pair of keys (that needs to be uploaded to your account on Git Hub). Please check online resources for the current Windows version you are using.

Pharo side settings

Then you need to go to Setting browser, search for "Use custom SSH keys" and complete your data there.

Alternatively, you can execute in your image playground:

```
IceCredentialsProvider useCustomSsh: true.
IceCredentialsProvider sshCredentials
  publicKey: 'path\to\ssh\id_rsa.pub';
  privateKey: 'path\to\ssh\id_rsa'
```

Pro Tip: this can be used too in case you have a non default key file, you just need to replace "id_rsa" with your file name.

8.5 Step 1. Setting up your repository

A fresh Pharo image comes with a pre-configured Pharo repository. Now this repository is in state "Local Repository Missing". You can develop with the image without problem now if you want to send some enhancements to Pharo you will not be able to do it. Indeed the message **Local Repository Missing** means that the image does not find a Pharo clone in your disk.

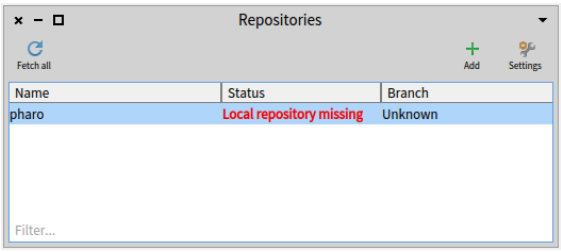

Figure 8-1 A fresh Pharo image by default indicates that it does not find the clone of Pharo.

Repairing local repository missing.

To solve this situation, you need to rebind your repository with a clone on your disk. Indeed even if you can browse the system class code, the image does not know where its repository is. The repair menu item/button will propose you some solutions.

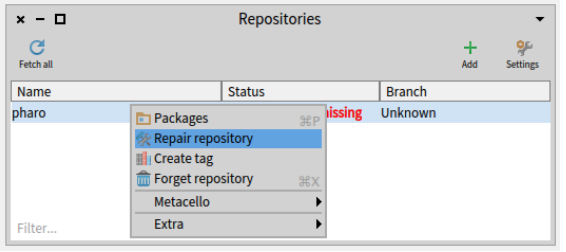

Figure 8-2 Repairing the Pharo project.

And clicking on the **Repair repository** menu will show you the repair view, showing an explanation of the current situation and some proposed solutions:

- Locate this repository in your file system: if you have already cloned your pharo fork repository on your disk you can simply point to it. Pharo will synchronize it as explained in the next steps.
- Clone again this repository: you can clone again your Pharo fork from GitHub to your local disc. Again your fork may be desynchronized from the actual Pharo version but Iceberg will manage it too.

So depending on the action you chose: you can then choose to search in your disk for an existing clone or to clone your forked Pharo repository.

Solving fetch required.

Once the repository is associated with its a clone, it will check the repository status and most probably you'll find out that your repository is in **Fetch required state**. Indeed "Fetch required" means that your image was built from a commit that cannot be found in your repository. In other words, we need to update your repository, doing a fetch it will update your repository.

To solve again, launch the repair action again. Usually, if you already have all your remotes correctly configured, doing a fetch will put you in a **Detached Working Copy** state. Pro Tip: Instead of doing a simple fetch (second option of the repair) it is quite powerful to create a branch (first option), give it a name like sync, bottom... This branch is pointing to the point where your repository and image are in sync and this is super useful to be able to check-

out it later when you want to do multiple PRs on different and unrelated issues. This way you can go back to a sync state in one click and be super productive.

Solving the Detached Working Copy.

Detached Working Copy means that the image commit does not correspond with the repository commit (more details of it in the Glossary). At this point, we need to synchronize both to be able to work.

Most of the time, the easier thing to do in this case is to just create a new branch as suggested in the Pro Tip before. If you plan to only fix on issue and you already know which issue you'll be working on, you can create a branch using the "New branch from issue" option. Otherwise, a nice alternative is to create a temporary branch like temp/synch that you can later on remove.

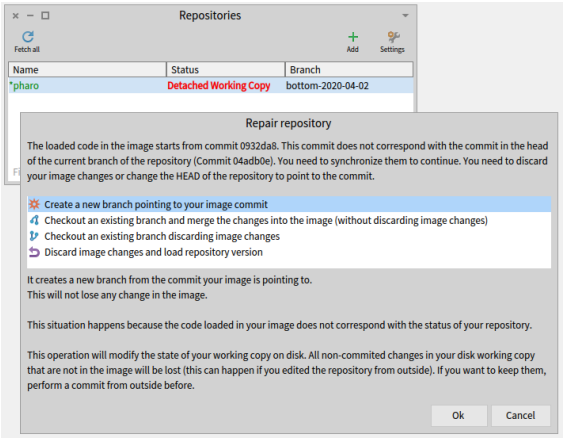

Figure 8-3 Solving the detached working copy situation.

8.6 Step 2: Work on your image and push your change

Pick up or create an issue and fix it. Once you have modifications in your image, it's time to push those changes to *your* fork and make a pull request from your fork to the main Pharo repository. To do that, we enforce the following process:

- you create a local branch for the issue,
- you issue a pull request of that issue to the main Pharo development branch.

Once you have done and committed all your work, you need to push it and create a pull request. Right click on the repository and go to the Github plugin menu item.

Select the option **Create pull request** and select as target branch Psssharo's development branch.

And issue the pull request! Pro Tip: If your PR fixes the issue, you can put a keyword "fixes #IssueNumber" in the description to automatically close the issue on merge.

8.7 Step 3: Follow your pull request

If you go to github, you'll see your pull request open and that some checks are running. Note that if your PR fails because of test failing or you want to integrate feedback provided by reviewed, it is super simple: just continue to commit in your branch. Your Pull Request will be updated with your new changes. This is always a cool feeling, so enjoy it.

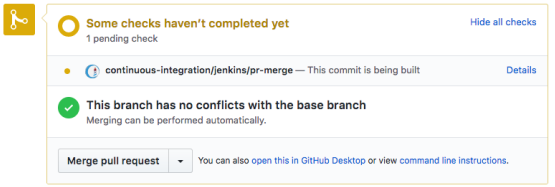

Figure 8-4 Checking your pull request.

8.8 Step 4: Once your pull request is integrated

Some cleanups are required:

- remove your branch from your fork
- close the issue (this is done automatically if your PR contained "fixes #numberOfIssue")

8.9 Why you do not need to resync your fork with the pharo repo?

No, you do not need to explicitly resync your fork. Why? Because when you push your commits, you push also the Pharo commits. But let us look at it in details:

- When you download an image (imagine that this image contains the code of commit3 and that Pharo' repository is at commit5), the repair action will fetch all the commits (up to 5) and put them in your *local* repository.
- When you create your bottom branch (the branch that points to the situation of your image when you downloaded) and later a child branch containing your fixes (call it myfix), it implies that when you commit your myfix branch, all the commits from myfixes and then from commit3 down to the common ancestor between your fork and your local repository will be pushed back to your *fork*.

So if you downloaded the latest image (imagine for commit6 and that the pharo repository is at commit6 - yes this is the latest image!) when you push your branch (which here is a child of commit6), automatically all the commits (6 down to the common ancestors between your local repo to your fork) will be pushed to your fork. So your fork will be in sync with Pharo's one.

So each time you work and push to your fork from the latest image, your fork gets updated!

8.10 Update your Pharo fork using the command line

You can also update your Pharo fork using the command line in 6 commands. You need to have a `git` command line interface installed:

```
$ git clone https://github.com/YOUR-USER-NAME/pharo.git
$ cd Pharo
$ git remote add pharo https://github.com/pharo-project/pharo.git
$ git fetch pharo
$ git pull pharo Pharo8.0
$ git push origin Pharo8.0
```

You can see the results in Figure 8-5.

8.11 The following script can be put in a ==.st== in your preferences folder (see below how to find it) and it will automatically configure Iceberg to connect to your accounts.

8.12 Conclusion

Now you know how to update a branch from your repository and can happily continue to improve Pharo.

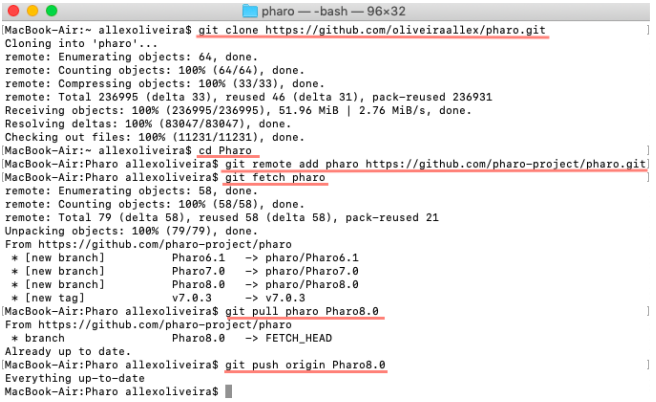

Figure 8-5 Command Line.

CHAPTER 9

Tips and Tricks

This chapter contains some simple recipe-oriented frequently asked questions. If you have some more please send them to us or even better, present a *Pull Request*.

9.1 How to use SSH keys

In case SSH is not set up (and you will notice as soon as you try to clone a project or commit a change to one), there are some simple basic points to be able to use SSH with github.

You can add SSH keys by following these steps (on Windows, if you want a nice command line environment, install http://mingw.org/wiki/msys):

Generate a key pair

To do this, execute the command:

```
ssh-keygen -t rsa
```

It will generate a private and a public key (on a unix-based installation in the directory `.ssh`). You should copy your `id_rsa.pub` key to your github account. Keep the keys in a safe place.

On Windows, follow instructions on how to generate your keys for your given version of Windows.

Add the key to your ssh agent

In linux, execute in your shell:

```
ssh-add ~/.ssh/id_rsa
```

In OSX, execute in your shell:

```
ssh-add -K ~/.ssh/id_rsa
```

For both OSX and linux you can add such lines to your `.bash_profile` (or the one corresponding to your shell installation such as `.zshrc`) so they are automatically executed on each new shell session.

9.2 Configuring automatically Pharo to commit in HTTP-S/SSH

With recent version of github, you should use a token to connect using HTTPS to be able to commit to your repository. Now you can configure Pharo to store your token as well your github account authentification using startup preferences.

In addition you can configure Pharo to point to your private and public SSH key as follows:

```
StartupPreferencesLoader default executeAtomicItems: {
    StartupAction
        name: 'Logo'
        code: [ PolymorphSystemSettings showDesktopLogo: false] .
    StartupAction
        name: 'Git Settings'
        code: [
            Iceberg enableMetacelloIntegration: true.
            IceCredentialStore current
                    storeCredential: (IcePlaintextCredentials new
                    username: 'xxJohnDoe';
                    password: 'xxJohnDoePassword';
                    host: 'github.com';
                    yourself).
            IceCredentialsProvider sshCredentials
                username: 'git';
                publicKey: 'Path to your public rsa.pub file (public
    key)';
                privateKey: 'Path to your private rsa file (private
    key)'.
            IceCredentialStore current
                storeCredential: (IceTokenCredentials new
                    username: 'xxJohnDoe';
                    token: 'magictoken here ';
                    yourself)
                forHostname: 'github.com'.
            ].
}
```

You should place this file in the preference folder that depends on your OS. You can find this place by using the `Startup` menu. Note that you can configure this for all or one specific version of Pharo.

9.3 How to contribute back to a project

You cloned a project from a Github repository and you don't have any write permissions. You made some changes, and you'd like to send a pull-request to it. What is the "Iceberg way" of doing it?

Manually you can fork the project on Github, add a remote to the local repo, push the changes to this new remote and then create the pull request on Github.

With Iceberg you can do the same since Iceberg follows the git way:

1. Create fork on Github
2. Add a repository for this project to Iceberg (Repositories context menu then "Repository" and then `"+ Repository"`
3. Create an issue branch with Iceberg (Repositories context menu then GitHub then Create new branch for issue).
4. Commit your changes with Iceberg.
5. Push your change to your fork from Iceberg.
6. Create pull request from Iceberg (Repositories context menu then GitHub then Create Pull Request)

9.4 How to distribute your changes in different branches

With Iceberg we can chose the packages, classes or methods when we save. Imagine that you are in branch `odd` and you code`m1`, `m2`, `m3`, `m4`. Then later you realize too late that it would be great to extract `m2` and `m4` in another branch because you would like to merge it in another branch called `even`.

To commit changes into different branches, you must change your branch. Because a *commit* always modifies the current branch.

When you change a branch, there is first a preview. In the preview, you have a diff and you have a combo-box to select a checkout strategy:

- The default checkout strategy is to reload the packages, this will override any local changes you had.
- The last one is to not load any packages: in other words, change the branch but do not touch my image.

So one way to do save your changes in separate branch is to:

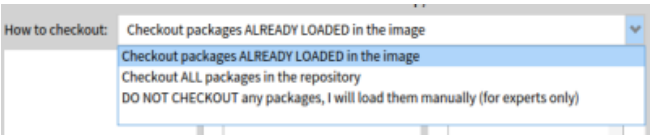

Figure 9-1 Checkout choices.

1. Commit in branch odd by selecting a subset of changes
2. Change to branch even using the ”Do not checkout” strategy
3. Commit in branch even a subset of changes

CHAPTER 10

Iceberg Glossary

Git is complicated. Git with (Pharo) images is even more complicated. This page introduces the vocabulary used by Iceberg. Part of this vocabulary is Git vocabulary, part of it is Github's vocabulary, part of it is introduced by Iceberg.

10.1 Git

Disk Working Copy (Git)

It is important not to confuse the code on your disk with the one of the repository itself. The repository (a kind of database) has a lot more information, such as known branches, history of commits, remote repositories, the git index and much more. Normally this information is kept in a directory named .git. The files that you see on your disk and that you edit are just a working copy of the contents in the repository.

The git index (Git)

The index is an intermediate structure which is used to select the contents that are going to be committed.

So, to commit changes to your local repository, two actions are needed:

1. `git add someFileOrDirectory` will add `someFileOrDirectory` to the index.
2. `git commit` will create a new commit out of the contents of the index, which will be added to your local repository and to the current branch.

When using Iceberg, you normally do not need to think about the index, Iceberg will handle it for you. However, you might need to be aware that the index is part of the git repository, so if you have other tools working with the same repository there might be conflicts between them.

Local and remote repositories (Git)

To work with Git you always need a local repository (which is different from the code you see on your disk, that is not the repository, that is just your working copy). Remember that the local repository is a kind of a code database.

Most frequently your local repository will be related with one remote repository which is called origin and will be the default target for pull and push.

Upstream (Git)

The upstream of a branch is a remote branch which is the default source when you pull and the default target when you push. Most probably it is a branch with the same name in your origin remote repository.

Commit-ish (Git)

A commit-ish is a reference that specifies a commit. Git command line tools usually accept several ways of specifying a commit, such as a branch or tag name, a SHA1 commit id, and several fatality-like combinations of symbols such as `HEAD^`, `@{u}` or `master\~2`.

The following table contains examples for each commit-ish expression. A complete description of the ways to specify a commit (and other git objects) can be found here.

Format	Examples
1. <sha1>	dae86e1950b1277e545cee180551750029cfe735
2. <describeOutput>	v1.7.4.2-679-g3bee7fb
3. <refname>	master, heads/master, refs/heads/master
4. <refname>@{<date>}	master@{yesterday}, HEAD@{5 minutes ago}
5. <refname>@{<n>}	master@{1}
6. @{<n>}	@{1}
7. @{-<n>}	@{-1}
8. <refname>@{upstream}	master@{upstream}, @{u}
9. <rev>^	HEAD^, v1.5.1^0
10. <rev>~<n>	master~3
11. <rev>^{<type>}	v0.99.8^{commit}
12. <rev>^{}	v0.99.8^{}
13. <rev>^{/<text>}	HEAD^{/fix nasty bug}

```
| 14. :/<text>              | :/fix nasty bug
-------------------------------------------------------------------
```

10.2 Iceberg

Iceberg Working Copy (Iceberg)

Iceberg also includes an object called the working copy that is not quite the same as Git's working copy. Iceberg's working copy represents the code loaded in the Pharo image, with the loaded commit and the packages.

Local Repository Missing (Iceberg)

The Local Repository Missing status is shown by iceberg when a project in the image does not find its repository on disk. This happens most probably because you have downloaded an image that somebody else created, or you deleted/moved a git repository in your disk. Most of the times this status is not shown because iceberg automatically manages disk repositories.

To recover from this status, you need to update your repository by cloning a new git repository or by configuring an existing repository on disk.

Fetch required. Unknown ... (Iceberg)

The Fetch required status is shown by Iceberg when a project in the image was loaded from a commit that cannot be found in its local repository. This happens most probably because you've downloaded an image that somebody else created, and/or your repository on disk is not up to date.

To recover from this status, you need to fetch from remotes to try to find the missing commit. It may happen that the missing commit is not in one of your configured remotes (even that nobody ever pushed it). In that case, the easiest solution is to discard your image changes and checkout an existing branch/commit.

Detached Working Copy (Iceberg)

The Detached working Copy status is shown by Iceberg when a project in the image was loaded from a commit does not correspond with the current commit on disk. This happens most probably because you've modified your repository from the command line.

To recover from this status, you need to align your repository with your working copy. Either you can

1. discard your image changes and load the repository commit,
2. checkout a new branch pointing to your working copy commit or

3. merge what is in the image into the current branch.

Detached HEAD (Git)

The Detached HEAD status means that the current repository on disk is not working on a branch but on a commit. From a git stand-point you can commit and continue working but your changes may get lost as the commit is not pointed to by any branch. From an Iceberg stand-point, we forbid commit in this state to avoid difficult to understand and repair situations. To recover from this status, you need to checkout a (new or existing) branch.

Bibliography