Manage Your Code with Iceberg

Stéphane Ducasse and Guillermo Polito

March 25, 2019

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

Contents

Illustrations ii

I Manage your code with Iceberg

1 Publishing your first project 5
1.1 For the impatient . . . 5
1.2 Iceberg setup . . . 6
1.3 Create a new project on github . . . 7
1.4 Add a new project to Iceberg . . . 8
1.5 [Optional but strongly suggested] Adding a src directory . . . 9
1.6 Repair to the rescue . . . 9
1.7 Create project metadata . . . 10
1.8 Adding and commiting your package . . . 10
1.9 Defining a BaselineOf . . . 13
1.10 Loading from an existing repository . . . 14
1.11 [Optional] Add a nice .gitignore file . . . 14

2 Empowering your projects 17
2.1 Adding Travis integration . . . 17
2.2 On windows . . . 18
2.3 Adding badges . . . 18

3 Understanding the architecture 21
3.1 Glimpse at the architecture . . . 21

4 Iceberg Glossary 23
4.1 Git . . . 23
4.2 Iceberg . . . 25

Bibliography 27

Illustrations

1-1 Use Custom SSH keys settings. 7
1-2 Create a new project. 7
1-3 Cloning new project. 8
1-4 Just after cloning an empty project. 9
1-5 Adding an src folder to a repository. 9
1-6 Create project metadata action and explanation. 10
1-7 Showing where the metadata will be saved and the format encodings. . . . 11
1-8 Details of metadata commit. 11
1-9 The package is clean, metadata are saved, but it is not loaded. 12
1-10 Adding a package to your project. 12
1-11 Publishing your committed changes. 13
1-12 With a Baseline. 14

3-1 Create a new project. 22

Part I

Manage your code with Iceberg

The authors want to thank Sean de Nigris and Stefan Eggermont for the reviews and copy-edit for the early version.

CHAPTER 1

Publishing your first project

In this chapter we explain how you can publish your project on github using Iceberg. We do not explain concepts like commit, push/pull, merging, or cloning. We thank Peter Uhnak for his first blog on publishing Pharo code on Github.

As git is distributed versioning system, you need a local clone of the repository. This is to this local repository that your changes will be commited to before being pushed to remote repositories. In general you commit to your local clone, and from there you push to remote repositories like github or gitlab, or that of your company. Iceberg will do all the operations and more for you.

1.1 For the impatient

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

- Create a project on github or any git-based platform.
- Configure Iceberg to use custom ssh keys
- Open Iceberg.
- Add a project (chose clone from ...).
- Optionally, in the cloned repository, create a directory named `src` on your file system using either the FileList or your command line.
- Open your project and add your packages (It is always good to add a baseline).
- Commit your project.

- Push it to your remote repository.

You are done. Now we can explain calmly.

1.2 Iceberg setup

To be able to commit to your git project, 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.

In case SSH is not setup (and you will notice as soon as you try to clone a project or commit a change to one), 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):

Generating 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, you can follow instructions on how to generate your keys at http://guides.beanstalkapp.com/version-control/git-on-windows.html#installing-ssh-keys.

Adding the key to your ssh

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.

Tell Pharo to use your keys

Go to settings browser, search for "Use custom SSH keys" and enter your data there as shown in Figure 1-1).

Alternatively, you can execute the following expressions in your image playground or add them to your Pharo system preference file:

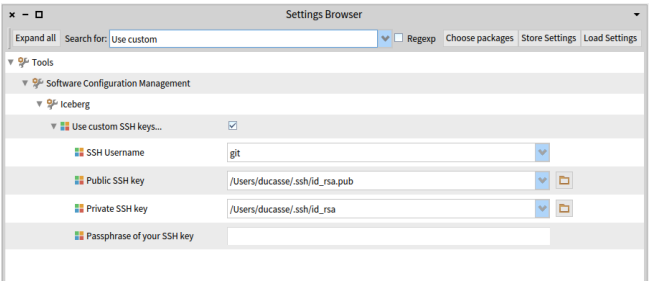

Figure 1-1 Use Custom SSH keys settings.

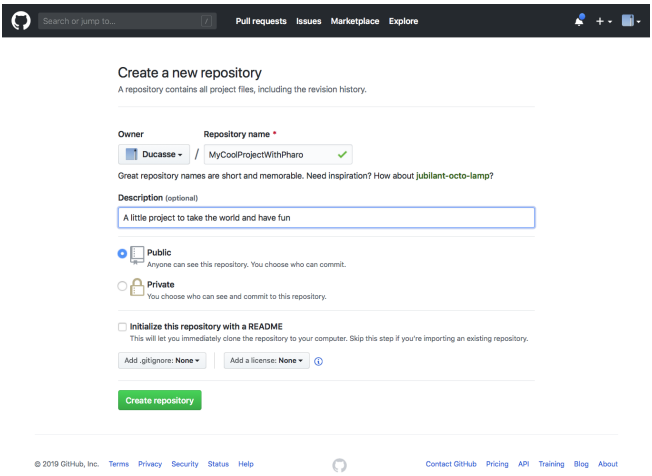

Figure 1-2 Create a new project.

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

> **Note** **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.

1.3 Create a new project on github

Figure 1-2 shows the creation of a project on Github.

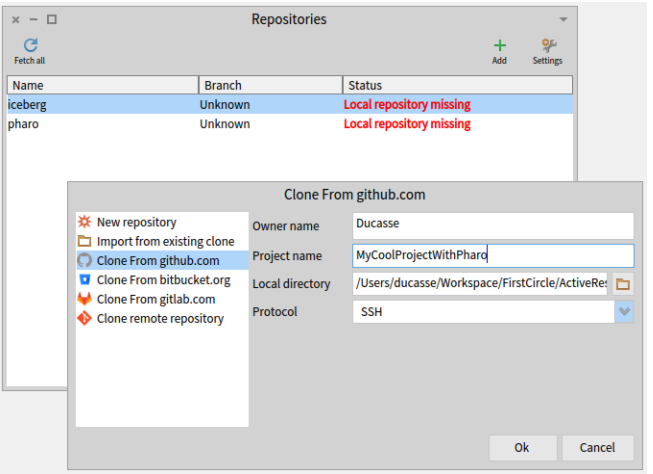

Figure 1-3 Cloning new project.

1.4 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.

Figure 1-3 shows that we are cloning the repository we just created. We specify the owner, project, and physical location where the local clone and git working copy will be on your disk.

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 1-4

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.

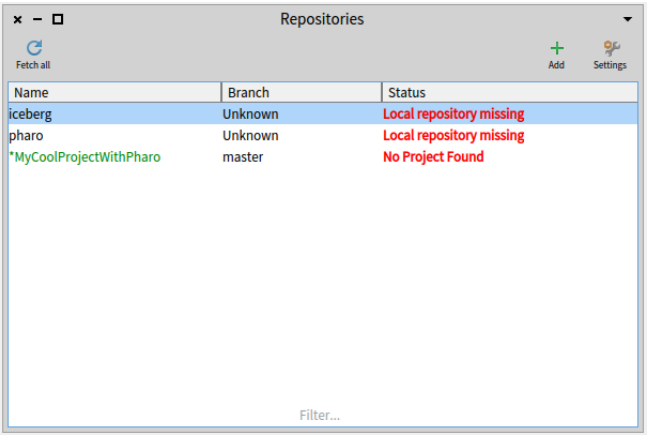

Figure 1-4 Just after cloning an empty project.

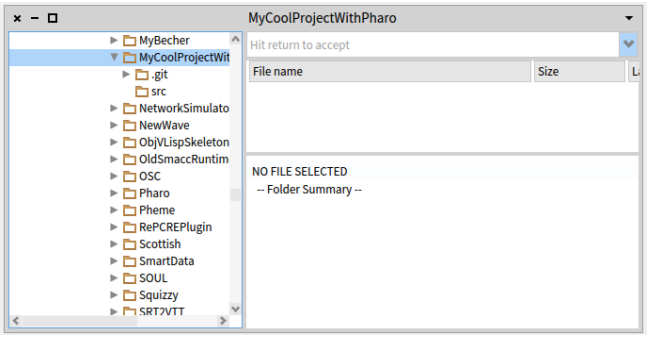

Figure 1-5 Adding an src folder to a repository.

1.5 [Optional but strongly suggested] Adding a src directory

Some developers like to group all their code in a directory named `src` and this is a nice practice. We strongly suggest that you follow it. You can go to your filesystem and create one in your repository. You can also use the Pharo FileList Browser to do it as shown in Figure 1-5.

1.6 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

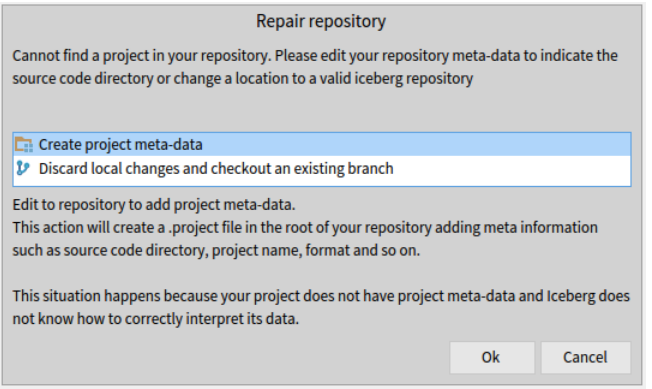

Figure 1-6 Create project metadata action and explanation.

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 probable. Each action will explain the situation and the consequence of the proposed action. It is always a good idea to read them.

Figure 1-6 shows the "Create project metadata" action and its explanation.

1.7 Create project metadata

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 1-7. 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're doing!

After accepting the project details, Iceberg shows you the files that you will be committing as shown in Figure 1-8

Once you have committed the metadata, Iceberg shows you that your project has been repaired but is not loaded as shown in Figure 1-9. This is normal since we haven't added any packages to our project yet. Your local repository is ready, so let's do that now.

1.8 Adding and commiting your package

Once your project contains Iceberg metadata, Iceberg will be able to manage it easily. Double click on your project and add a package by pressing the + (Add Package) button as shown by Figure 1-10.

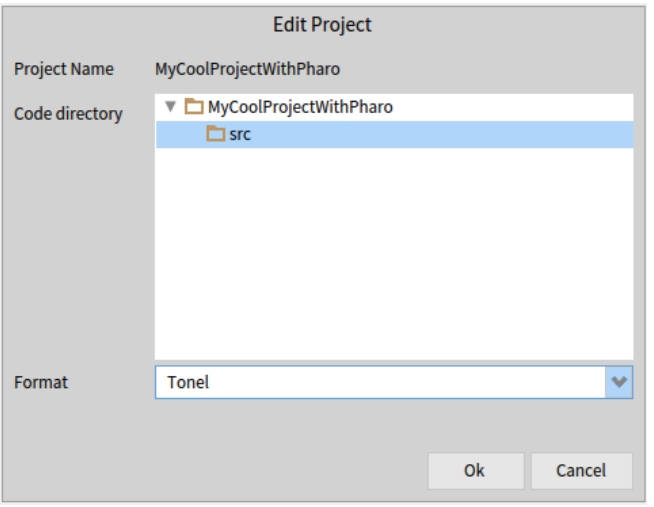

Figure 1-7 Showing where the metadata will be saved and the format encodings.

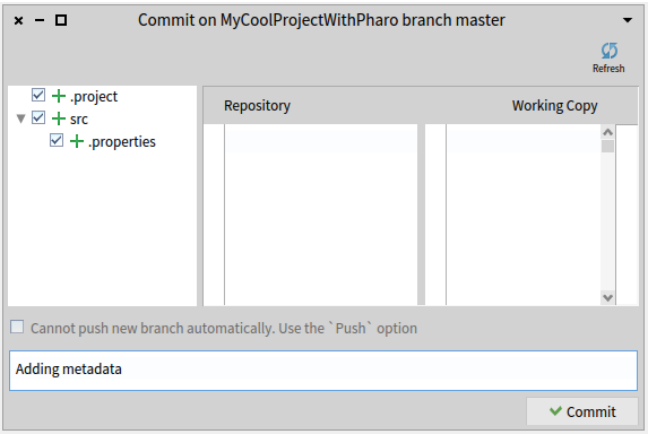

Figure 1-8 Details of metadata commit.

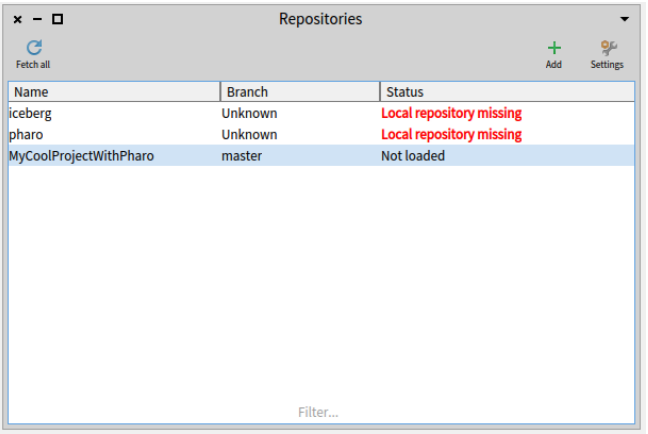

Figure 1-9 The package is clean, metadata are saved, but it is not loaded.

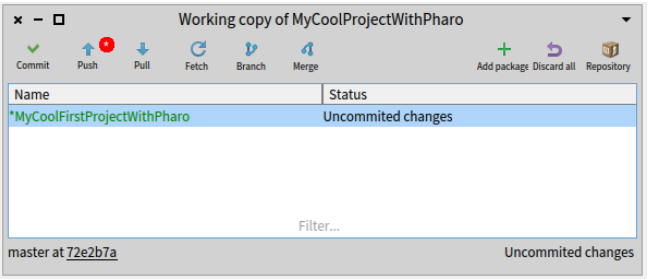

Figure 1-10 Adding a package to your project.

Again, Iceberg shows that your package contains changes that are not committed using the green color and the * in front of the package name.

Now you are left with two actions:

- Commit the changes to your local repository using the Commit button. Iceberg will reflect this change by removing the * and the changing the color.

You can commit several times if needed.

- Publish your changes from your local directory to your remote repository using the Push button.

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 1-11. 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.

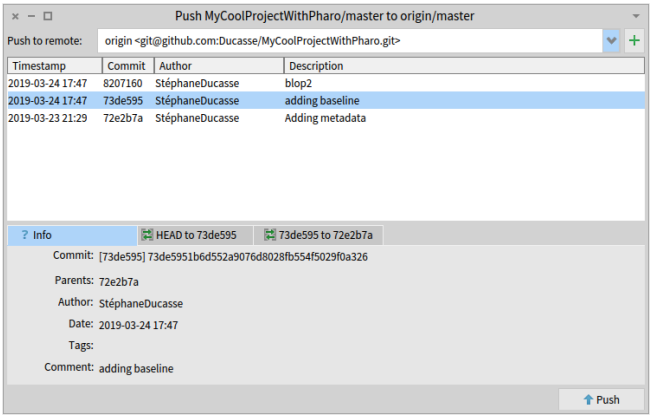

Figure 1-11 Publishing your committed changes.

1.9 Defining a BaselineOf

A BaselineOf is a description of a project's architecture. 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 and bother about 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: 'BMyCoolProjectWithPharo'.
      spec group: 'default' with: #('FileDialog') ]
```

Once you have defined your baseline, you should add it to your project as shown in Figure 1-12. Now, commit it and push your changes to your remote repository.

The online baseline documentation is available at: https://github.com/pharo-open-document pharo-wiki/blob/master/General/Baselines.md.

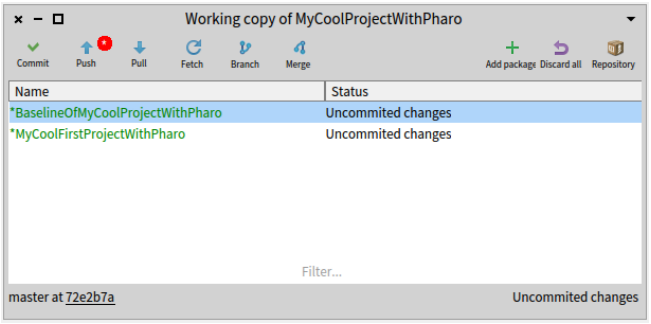

Figure 1-12 With a Baseline.

1.10 Loading from an existing repository

If you already have a repository, and you just want to load it into Pharo, there are two ways to go about it. The first is as we did above. You can select a package and manually load it.

The second makes use of Metacello. However, this will only work if you have already created a `BaselineOf`. In this case, you can just do:

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

For projects with metadata, like the one we just created, that's it. However, if you are loading a project without metadata, you must add the code subfolder to the end of the repository string i.e. 'github://Ducasse/MyCoolProjectWithPharo/src'.

1.11 [Optional] Add a nice .gitignore file

Iceberg automatically manages such files.

```
# 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
```

1.11 [Optional] Add a nice .gitignore file

```
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
```

CHAPTER 2

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 using Travis.

2.1 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. Check your travis account.

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've done everything right, Travis will pick up the changes and will start testing and building it... and you're done, congratulations!

2.2 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"'
```

2.3 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.

```smalltalk
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.

CHAPTER 3

Understanding the architecture

3.1 Glimpse at the architecture

As git is distributed versioning, you will need a local clone of your repository. You commit to your local clone, and from there you push to remote repositories like github.

Figure 3-1 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 git local 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 resynchronisation of your image with the git local repository, git remote repositories and the git working copy.

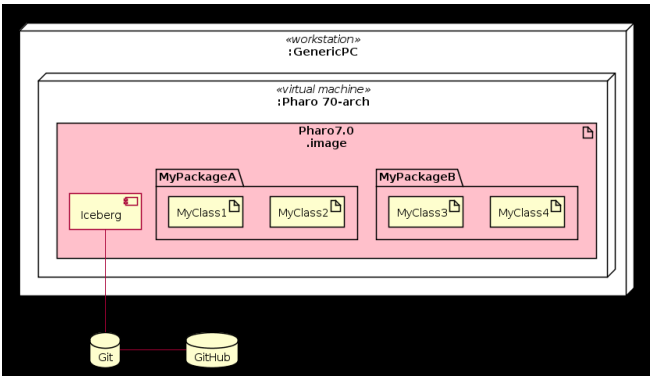

Figure 3-1 Create a new project.

CHAPTER 4

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.

4.1 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. Still, 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 database (for code).

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 at https://mirrors.edge.kernel.org/pub/software/scm/git/docs/gitrevisions.html#_specifying_revisions.

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
-------------------------------------------------------------------------
```

4.2 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've 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