

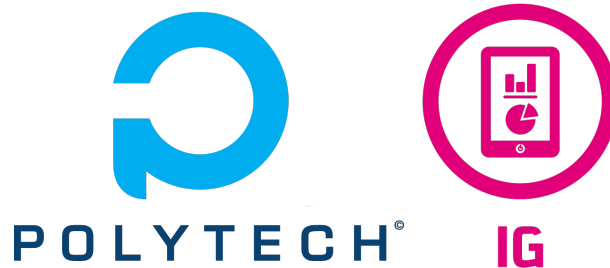
FUNCTIONNAL PROGRAMMING PROJECT
IT & MANAGEMENT DEPARTMENT

Creation of a Git-like tool in Scala

Author

Thomas FALCONE

October 20, 2019




Contents

1	Instructions	1
2	Architecture	1
2.1	Presentation	1
2.2	File organisation	2
2.3	Git copycat	2
2.3.1	Diff algorithm	3
2.3.2	Merge algorithm	3
2.4	Pros	4
2.5	Cons	4
3	Tests	4
3.1	Naïve approach	4
3.2	I-understood-the-mistake approach	4
4	Post-mortem	4
4.1	Teamwork has a lot of benefits	4
4.2	Testability is important	4
4.3	Dev tools can be a great guiding light	4

Every link of this document is clickable if you are reading this in a digital format

1 Instructions

You can find the source code on Github through the following link : Source code 
<https://github.com/ThomasF34/sgit>

In order to install **sgit** you can either use the Github release TODO or build the binary file from the source code. *Be aware that you must have installed sbt before trying to compile the source code. You will find more information about sbt on this site <https://www.scala-sbt.org/>*

To compile from source code use :

```
1 sbt assembly
```

Then add the file *target/scala-2.13/sgit* to your PATH.

To install via the Github release, simply download the binary file and add it to your PATH.

*You are now able to use sgit. You can find the usage with **sgit -help***

2 Architecture

2.1 Presentation

The architecture of sgit is quite explicit and can be seen as a simple flow.



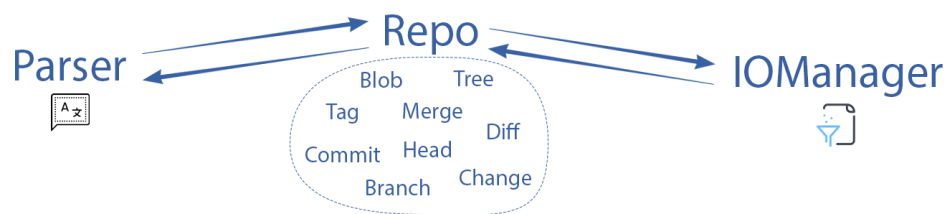
Everything begins within the **Parser** that will take the given command, initiate the **Repo** instance and ask it to execute the wanted action. It will then load all the needed the content from the **IOManager**, the HQ of File management and compute the result.

In order to achieve this goal, the Repo uses an abstraction of the object it manipulates. Hence, it exists several *case classes* that will help readability and separate each other's concerns.

Here is a list of existing classes :

- Blob
- Tree
- Branch
- Tag
- Commit
- Head
- Merge
- Diff
- Change

Theses objects are using High Order Function in order to gain access to the content they need. For example we give a function called *commitContent* to the Commit object so he can access some commit's information. Hence, every function that need to have a commit information respects Referencial Transparency with this function injection. The Repo instance is responsible for this function injection. It is using IOManager's function to let other object access file's content, while hiding its origin to the aforementioned object.



Also, I used currying in order to specialize theses functions we are giving to other classes. Let's see how I used currying with an example :

```

1 def getContent(dir: String)(filename: String): String = {
2   val file = new File(s"${dir}${filename}")
3   if (file.exists() && file.isFile()) {
4     Source.fromFile(file).mkString
5   } else ""
6 }

```

Listing 1: Function in IOManager

```

1 lazy val blobContent = ioManager.getContent(blobsPath)(_)
2 lazy val tagContent = ioManager.getContent(tagsPath)(_)

```

Listing 2: Functions to be injected

With these functions, the Blob instance (to which we are gonna inject *blobContent*) will only be able to access blob content while Tag instance will only be able to access tag content.

2.2 File organisation

2.3 Git copycat

In order to reproduce most of Git function I had to implement some interesting algorithms.

First for the *diff* functions and then for the *merge* one.

2.3.1 Diff algorithm

In order to obtain the diff between two files I implemented an algorithm base on the Longest Common Sequence (LCS) between two collections (here two collection of string where each element represents a line of the compared file)

2.3.2 Merge algorithm

To compute the merge algorithm I had to obtain the LCS between three files. The first and the second one are the files on each branch we want to merge and the last one is the file on the common ancestor commit.

I implemented an algorithm to compute the LCS matrix 'in 3D' and even if it sounds more complex, it is not really the case. In fact, for the merge algorithm we don't need to know if each modified line had been inserted or deleted.

The complex part of this algorithm, though, is *the align* step. I had to align all three file based on their LCS to then compare line by line and apply changes.

Here is an imaged example of what the align step will do. (You can retrieve the algorithm implementation in the *Merge.scala* file ¹)

A	A	A
B	C	B
C	D	C
D	F	E
E		F
F		

(a) Before alignment

A	A	A
B	∅	B
C	C	C
D	∅	∅
E	∅	E
F	F	F

(b) We can now compare line by line

¹<https://github.com/ThomasF34/sgit/blob/master/src/main/scala/igpolytech/Merge.scala>

2.4 Pros

2.5 Cons

3 Tests

3.1 Naïve approach

3.2 I-understood-the-mistake approach

4 Post-mortem

4.1 Teamwork has a lot of benefits

4.2 Testability is important

4.3 Dev tools can be a great guiding light