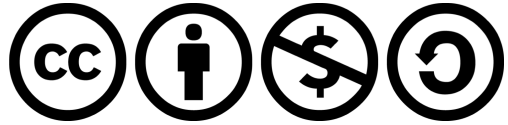
Version Control I

Introduction to git – git-local

Programming Practices for Economics Research

Department of Economics, Univeristy of Zurich

Fall 2017



Learning Objectives

- At the end of the session you will be able to:
  1. Convey the advantages of Version Control Systems
  2. Understand the vocabulary of Git
  3. Work with Git on your computer
  4. Use branches and merge work streams
  5. Know where to read up advanced stuff

Why Git?

The problem:

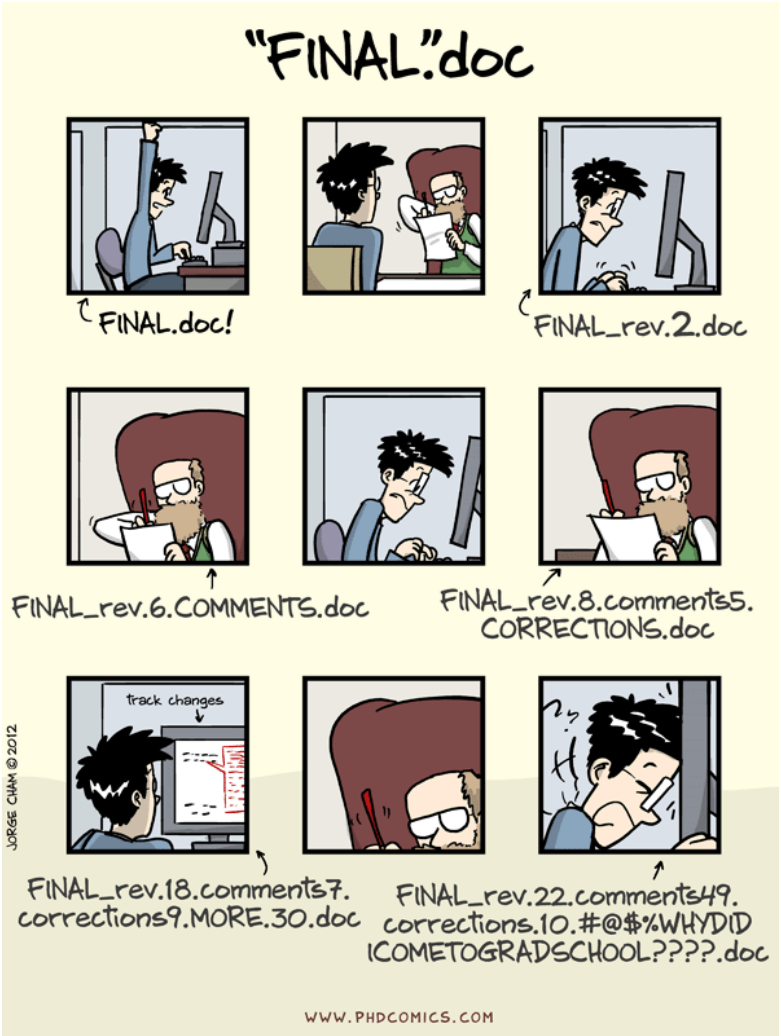


Figure 1: Final Doc

Ad-hoc solution: Save stuff regularly

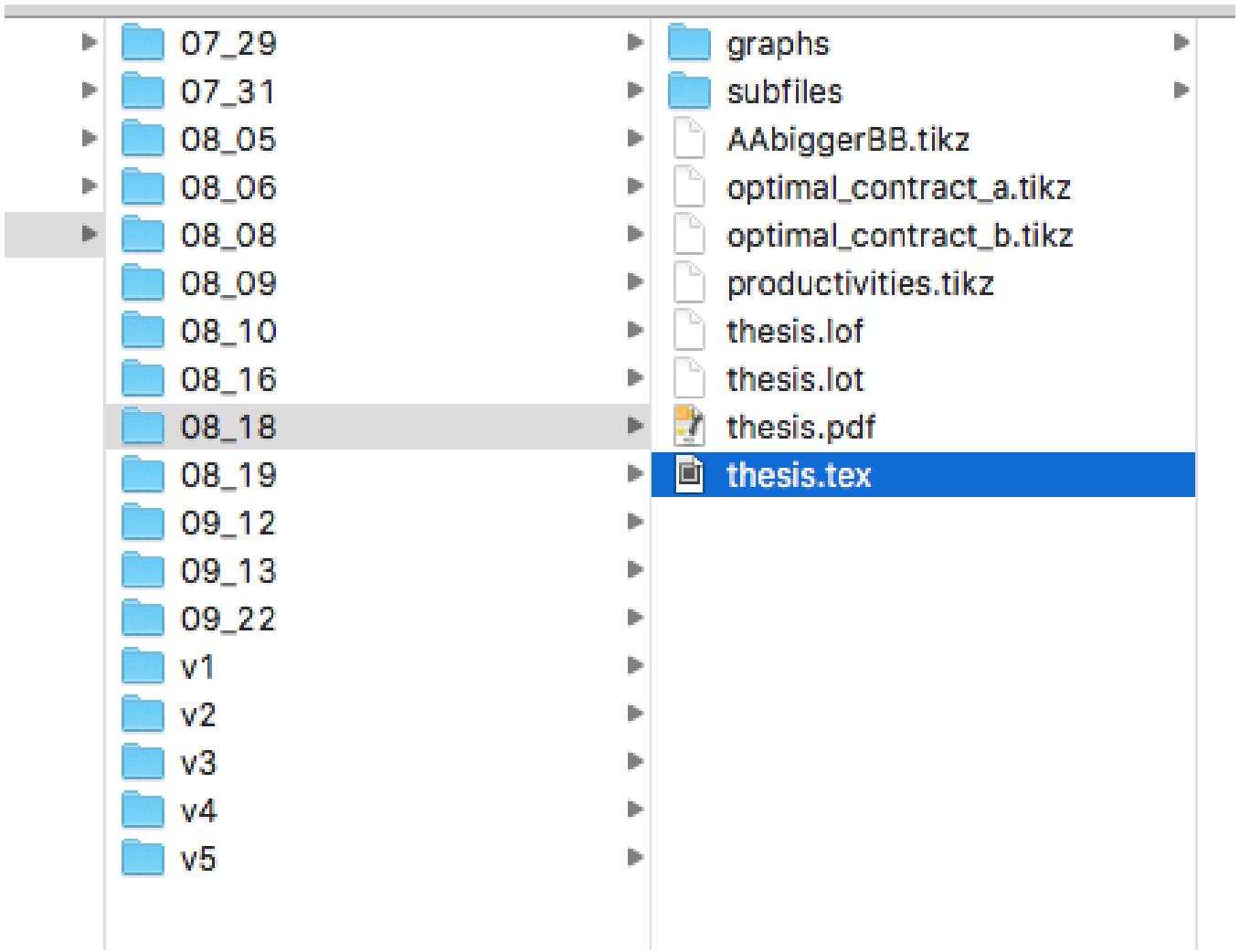


Figure 2: Where was the last good version of that paragraph?

The better solution: Git

464 LaTeX/research-plan.tex View

```
@@ -1,266 +1,238 @@
 \section{Summary of the research plan}

- Almost all economic research assumes and relies on stable and rational preferences, which are necessary for the existence of a utility function -- the most basic and uncontroversial building block of our profession. Observed choice data has to necessarily satisfy the Weak Axiom of Revelead Preferences (WARP), which implies that a choice option can never become \emph{more} popular if the set of choices increases. Similarly, models of the random utility family, our workhorse models to analyze discrete choice data empirically, like the Logit, Probit, Multinomial Logit, Mixed Logit etc, assume that the choice frequency of a given option can \emph{never} be higher in a larger choice set (cite Marshak). In fact no data can be generated by a random utility process which violates this regularity assumption (cite Natenzon 2017). These models also perform poorly when analyzing such data.

- Experimental researcher are able to generate such a violation for more than 30 years. Starting with the discovery of the attraction effect by (cite 1982) and the compromise effect by Simonson (cite 1989), more than 100 studies were able to replicate this anomaly. To illustrate the attraction effect, let us start with a choice set with two items: a cashmere sweater which costs \$100 and a synthetic one with costs \$20. The attraction occurs when we add a new choice option, a cotton sweater which costs \$110, to the original choice set. This new option, the decoy, is dominated in both dimensions by the cashmere sweater as it is more expensive and of a lesser material in comparison to the cashmere sweater. At the same time it is better only in the material dimension but more expensive than the synthetic sweater. While a price conscious consumer might prefer the cheap, synthetic option in the original choice set, experimental and field evidence shows that the addition of the decoy will reverse this preference for many consumers. This effect is especially intriguing to the marketing profession. As the dominated option is never chosen by consumers and increases profits without changing prices or product attributes of the original choice options, it has gained a lot of attention in the marketing literature and business practice. We know that the Economist uses attraction effect pricing to nudge consumers into buying the priciest 'print + online' subscription (cite Ariely predictably irrational). (cite blabla) also found in a field setting that consumers of a supermarket can be nudged into buying the pricier option of canned beens when a dominated option is added to the menu.

 \section{Summary of the research plan}

+ Almost all economic research assumes and relies on stable and rational preferences, which are necessary for the existence of a utility function -- the most basic and uncontroversial building block of our profession. Observed choice data has to necessarily satisfy the Weak Axiom of Revelead Preferences (WARP), which implies that a choice option can never become \emph{more} popular if the set of choices increases. Similarly, models of the random utility family, our workhorse models to analyze discrete choice data empirically, like the Logit, Probit, Multinomial Logit, Mixed Logit etc, assume that the choice frequency of a given option can \emph{never} be higher in a larger choice set (cite Marshak). In fact no data can be generated by a random utility process which violates this regularity assumption Consequently, random utility models perform poorly in the analysis of such data giving estimate that do not reflect the regularity violation (cite Natenzon).

+ Experimental researcher are able to generate such violations for more than 30 years with what is now know as the decoy effect. The decoy effect can be divided into two related effect, the attraction effect which was discovered in 1982 by (cite 1982) and the compromise effect which was discovered by Simonson in 1989 (cite 1989). Since then, more than 100 studies were able to replicate the anomalies. To illustrate the \emph{attraction effect}, let us start with a choice set which contains two items: the target, a high quality cashmere sweater which costs \$100 and the competitor, a low quality synthetic one with costs \$20. A price conscious consumer might prefer the cheaper option in this original choice set. The attraction occurs when a third option, the decoy, which is dominated in price and quality only by the target, is added to the choice set. A typical decoy in this example is a cotton sweater which costs \$110. Both survey and field data show that the introduction of the decoy will make the high quality target look like a good deal in comparison to the low quality competitor and move consumers into purchasing the target more frequently. This effect is especially intriguing to the marketing profession. As the dominated option is never chosen by consumers and increases profits without changing prices or product attributes of the original choice options, it has gained a lot of attention in the marketing literature and business practice. We know that the Economist uses attraction effect pricing to nudge consumers into buying the priciest 'print + online' subscription (cite Ariely predictably irrational). (cite blabla) also found in a field setting that consumers of a supermarket can be nudged into buying the pricier option of canned beens when a dominated option is added to the menu.
```

Figure 3: Here it is

Git: Vocabulary

- Git is a Version Control System.
- **Repository:** Contains the whole history of a project from its inception
- **Commit:** A snapshot of a project at a certain point in time including all the files and directories
- **Commit Graph:** Shows how all commits are related to each other (parents/children etc.)
- **Head:** The latest snapshot at the end of a graph
- **Branches:** At the same time you can have several versions of a project. You can work on all of them parallel and **Merge** them together once their done (e.g. Master, Experimental, Feature X)

Git: Tracking Changes

The idea

Use Git

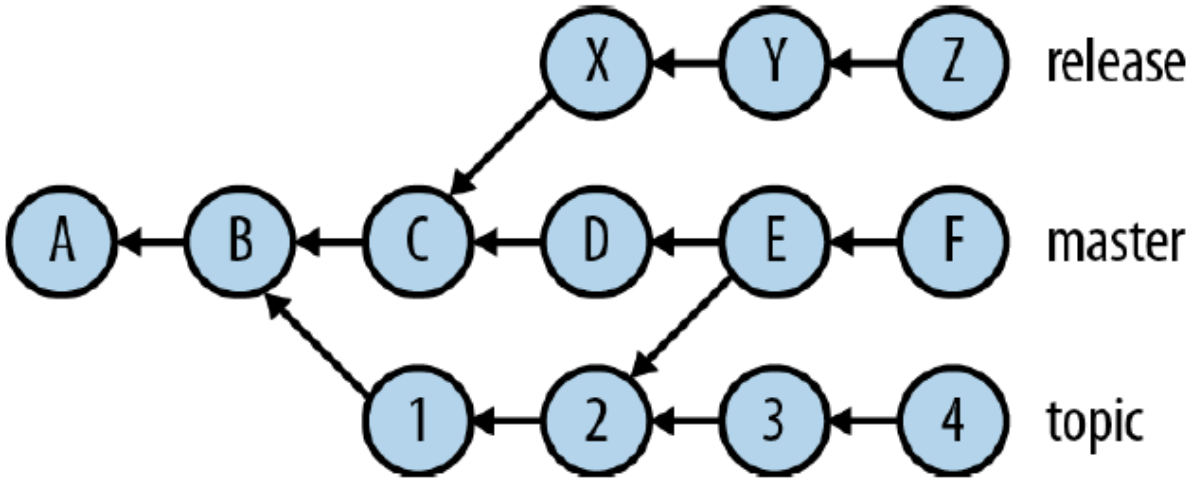


Figure 4: Commit Graph

Use Git

- The labels on the right side of the previous picture —- master, topic, and release —- denote *branches.*
- The branch is defined as the collection of all commits that are reachable from the tip – the latest commit of a branch – by following the parent arrows backward along the history.

Git: Best practice

Private

- When working on your own, it's useful to commit *early and often*, so that you can explore different ideas and make changes freely without worrying about recovering earlier work.

Public

- Once you go public, it's important that others will understand your progress and your commits.

How to use Git

On windows

- Use cygwin

On Mac/Linux

- use your terminal/bash

On both

- Use built-in Atom functions
- Use your terminal in Atom

Set up your Git Credentials

Eventually, a big advantage of Git will be that it tracks who did what and when

So first tell Git who you are and how you may be reached

- `$ git config --global user.name "First Last"`
- `$ git config --global user.email "first.last@econ.uzh.ch"`
- use the `--local` option if you want to define a different name or an email address for the current repository.

Change your standard editor for commit messages from VIM

- NANO: git config –global core.editor “nano -w”
- ATOM: git config –global core.editor “atom –wait”

Hands-on

The Basic Workflow

Creating a New Local Repository

- `$ git init`
  - create a new **.git** directory in the current working directory
- `$ git status`

The Index or Staging Area

- move a file your working directory
- `$ git add [somefile]`
  - add some file to the Git index
- `$ git status`

Importing an Existing Project

- `$ git init`
- `$ git add .`
- `$ git status`

In action...

- Create your first Git repository
- create a new file using `atom newfile.txt`, write something and save it
- type `git status`
- add it to the index
- confirm with `git status`

Keep Only Source Files under Version Control

- An output file, i.e., a pdf file, changes every time you recreate the file, even if there are no material changes to the file
  - there will be many fake changes of the repository
  - the repository size will explode
- Hence, keep only sources under VC!
  - Original data and source code from statistics programs, LATEX sources, etc.

Ignoring Output and Nuisance Files

- In large projects, it become impossible to manually select files to be added to the index.
- Hence, specify patterns to be ignored in a file called `.gitignore`, which lives in the project root.
- Use template `.gitignore` files
  - `$ git add .gitignore -f`
- It is still possible to manually add files that are ignored.

The First Commit

- Commit to the local repository with a meaningful message
- `$ git commit -m "Initial commit."`
- `$ git status`

Let's Do It

- Make Your First Commit

The Second Commit

- Work on the files you added to the index.
- `$ git commit -am "Changes XZY."`
  - commit all changes to the local repository. the -a option adds all tracked, modified files to the index before committing and commits changed and deleted files, but not new ones.
- use short and meaningful messages.
- `$ git status`

The Third Commit

- Make some changes in your file
- type `$ git diff` to see the changes you made since your last commit
- if you are happy, commit your progress

Changing The Index

- `$ git add [filename]`
- check with `$ git diff --staged`
- `$ git commit -am "Changes XZY."`
- `$ git status`

Changing The Index

- `$ git add -u`
  - include all files in the current index, except new ones
- `$ git add -A`
  - include all files in the working tree, including new files.
- `$ git rm [filename]`
  - delete the file from the index **and delete the working file**
- `$ git mv [oldname] [newname]`
  - rename the file
- `$ git reset`
  - reset the index to match the current commit
- `$ git commit -am "Changes XZY." --amend`
  - discard the previous commit and put a new one in its place to include new files (-a does not include new files).
- `$ git status`

View the Log of Commits

- `$ git log`
  - show the history of commits
- `$ git log -g`
  - shows the history of operations, including ammended commits

Discarding the Last Commit

- `$ git reset HEAD~`
  - move the branch back to one commit, discarding the latest one
  - you can still recover the latest one using `$ git log -g`
- `$ git reset HEAD~3`
  - discard any number of consecutive commits; here, go back to the fourth commit (0 is the current commit)

Undoing Commit

- `$ git revert [HASH]`
  - use `git log` to get the HASH
  - make a new commit undoing the earlier commit's change
  - you can still recover the latest one using `$ git log -g`

Restore an Old Commit

- `$ git checkout HEAD [yourfile.txt]`
  - recovers the last saved commit
- `$ git checkout [HASH] [yourfile.txt]`
  - recovers any previous commmit according to its hash. Recover the commit number that captures the state of your repository *before* the change you are trying to undo.

NOTE: if you forget `[yourfile.txt]` in that command, git will tell you that "You are in 'detached HEAD' state." In this state, you shouldn't make any changes. You can fix this by reattaching your head using `git checkout master`.

Let's Do It

- Make some changes to your file. Use `$ git diff`
- Add more files to the index
- Make some commits
- Check out the history of your commits
- move back and forth on your branch
- undo some changes

Branching

The Default Branch, master

- Branches allow different versions of the same content to evolve independently at the same time, while you periodically recombine the contributions from different branches in a process called **merging.**
- when you create a new repository, typing `$ git init`, you start on the master branch by default

The Commit Graph Again

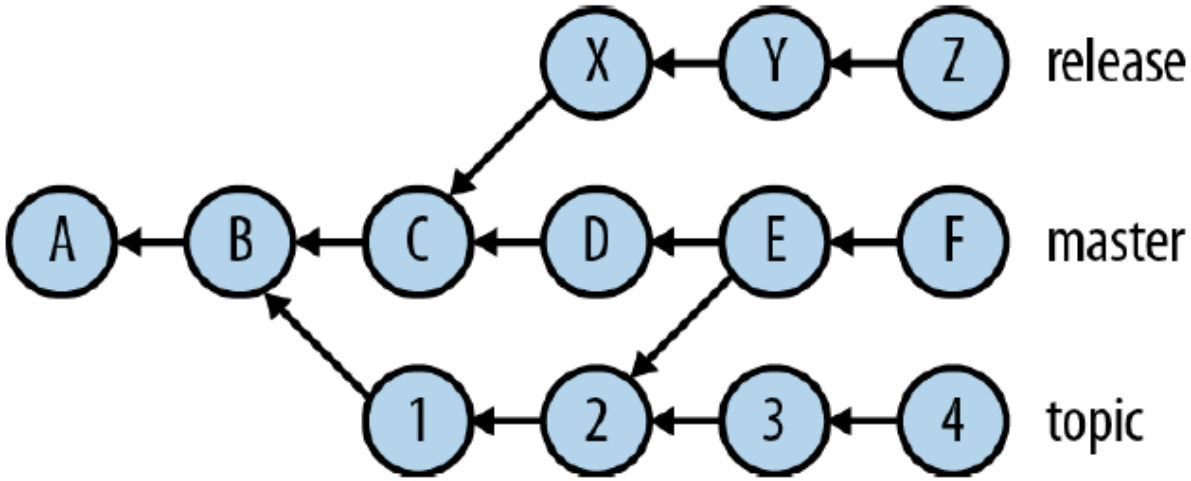


Figure 5: Commit Graph

Making a New Branch

- `$ git checkout -b experimental`
  - create a new branch *experimental* pointing at the current commit, and switch to it.
- `$ git checkout -b experimental [HEAD]`
  - start a new branch at the commit named by HEAD and switch to it

Switching Branches

- `$ git branch`
  - have an overview of all local branches
- `$ git checkout [branchname]`
  - switch to a different branch from the repository and work with it
- `$ git checkout master`
  - move back to the master branch
- use GitX (Mac) or Tortoise (Windows) to have the overview of different branches and changes

Deleting a Branch

- `$ git branch -d [branchname]`
  - delete the branche
- `$ git branch -D [branchname]`
  - force to delete the branche

Let's Do It

- Create a new Branch on which you play with your file
- Make some Commits of your changes
- Move back and forth between your branches

Merging

Merging

- Merging is the process of combining the recent changes from several branches into a single new commit that is on all those branches
- A typical work flow looks like this:
  - `$ git checkout -b experimental`
  - `$ git commit -am "some brilliant change"`
  - `$ git checkout master`
  - `$ git status`
  - `$ git merge experimental`
  - `$ git commit -am "merged"`
  - `$ git status`

Merge Conflicts

- If there are files with conflicts Git could not resolve, use `$ git diff` to find out what went wrong.
- once you have edited the file to resolve the conflict, use `$ git add` to stage your fixed version for commit and remove it
- once you have addressed all the conflicts, `$ git status` should no longer report any unmerged paths.
- complete the merge with `$ git commit`

Resolving Merge Conflicts

- `$ git log -p --merge` shows all commits containing changes relevant to any unmerged files together with their diffs.
- If you want to discard all the changes from one side of the merge, use `$ git checkout --{ours,theirs} [file]` to update the working file with the copy from the current or other branch, followed by `$ git add [file]` to stage the change and mark the conflict as resolved.
- Having done that, if you would like to apply *some* of the changes from the opposite side, use `$ git checkout -p [branch] [file]`.
- complete the merge with `$ git commit`
- we will do some exercise on this in the next lecture

Tagging Helps You to Find Specific Versions

- `$ git log`
- `$ git checkout [HEAD]`
  - move to a specific commit using the hash
- `$ git tag meaningful_tag -m "An interesting message"`
- `$ git checkout master`
- `$ git tag`
- `$ git checkout [tag]`

In Action...

- create a new branch and do some changes to your file, commit them.
- go back to your master branch and do some conflicting changes, commit them
- merge the new branch to you master branch
- solve the merge conflicts
- tag your final version with a cool tag and an interesting message

Some Final Remarks

- When everything stops working. . .
- . . . don't panic!!!
  - Situation from the last commit is always in the repository
  - So be sure to commit frequently
  - Always solve problems immediately so that you won't loose much information should you have to go back
- Won't happen much now – but things become a bit tricky once we use Git for collaboration

Where to Find Help

- Here are two good books to look up stuff:
  - Loeliger and McCullough (2012)
  - Silverman (2013)

What you should have taken away...

1. why is it cool to use Git?
2. do you understand the vocabulary: repository, branches, commits?
3. can you track your own work?
4. can you set up new branches to experiment and merge it to your master when you are happy?
5. Do you know where to look up stuff if you want to know more?

Acknowledgements

- This course is designed after and borrows a lot from:
  - Effective Programming Practices for Economists, a course by Hans-Martin von Gaudecker
  - Software Carpentry and Data Carpentry designed by Greg Wilson
  - Shotts, W.E. (2012). The Linux Command Line. San Francisco: No Starch Press.
- The course material from above sources is made available under a Creative Commons Attribution License, as is this courses material.

Programming Practices Team

Programming Practices for Economics Research was created by

* Lachlan Deer
* Adrian Etter
* Julian Langer
* Max Winkler

at the Department of Economics, University of Zurich. These slides are from the 2017 edition.

Loeliger, Jon, and Matthew McCullough. 2012. *Version Control with Git: Powerful Tools and Techniques for Collaborative Software Development*. " O'Reilly Media, Inc.".

Silverman, Richard E. 2013. *Git Pocket Guide*. " O'Reilly Media, Inc.".