# Version Control with Git

## Working on a Single Machine

Programming Practices for Economics Research

Department of Economics, Univeristy of Zurich

Winter 2019

# 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

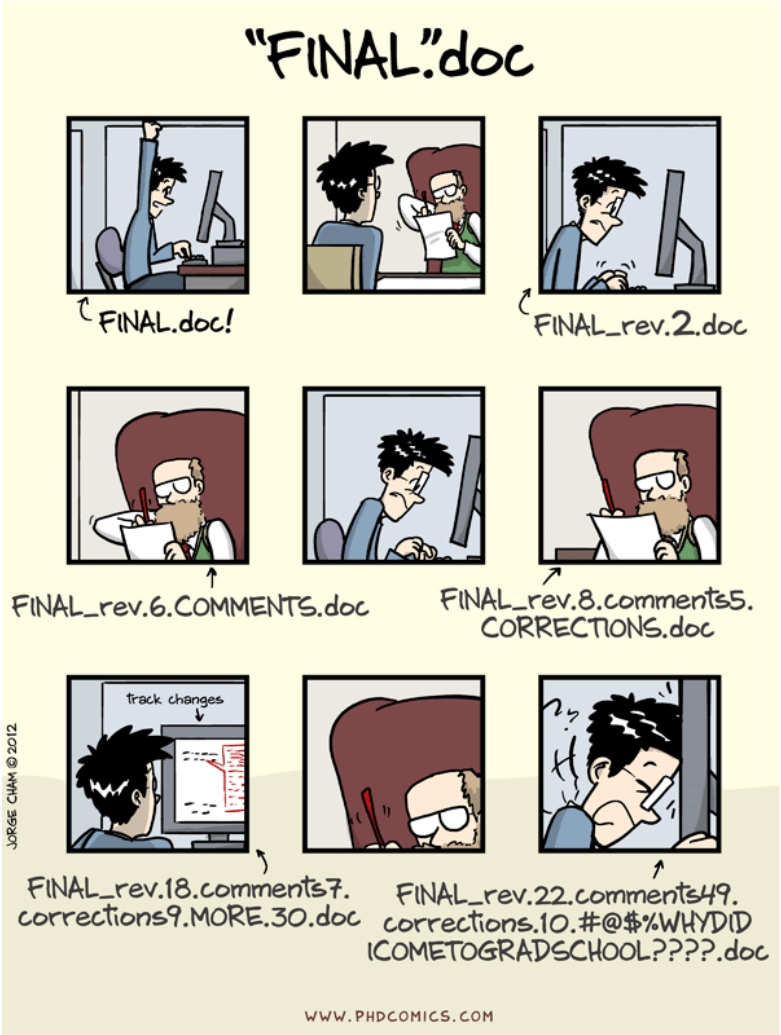


**Figure 1:** Final Doc

# Ad-hoc solution: Save stuff regularly

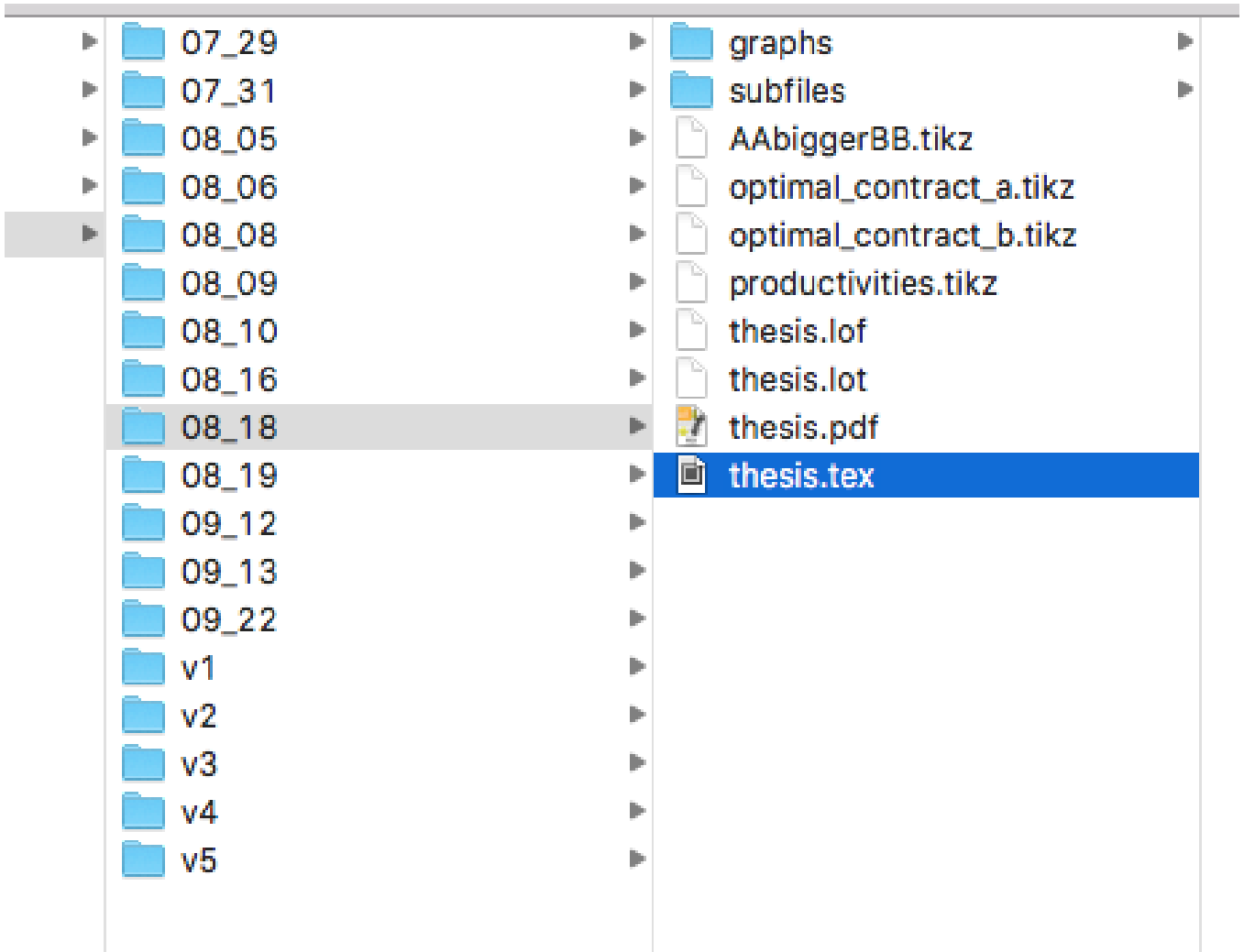


**Figure 2:** Where was the last good version of that paragraph?

# 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 Tracks Differences in Files

```
diff --git a/somefile.ext b/somefile.ext
index 6ff6786..54994ef 100644
---- a/somefile.ext
+++ b/somefile.ext
@@ -1,16 +1,16 @@

+ I
+ added
+ three lines

- i removed
- another
- four
- lines
```

# Git logs changes to create a history

```
commit ce81835860915590 9ba0e22bc7906499125bbb6a
Author: Joe Bloggs <joe.blogggs@gmail.com>
Date:   Wed Feb 20 20:44:44 2019 +0100

    Rewrote motivation paragraph in the intro to match
    new results

commit 8441f3e1017a714f59830a7d033e561f6c402782
Author: Fred Smith <fred.smith@gmail.com>
Date:   Tue Feb 19 13:14:28 2019 +0100

    Added Geography controls to main regressions
```

# Project history becomes a DAG

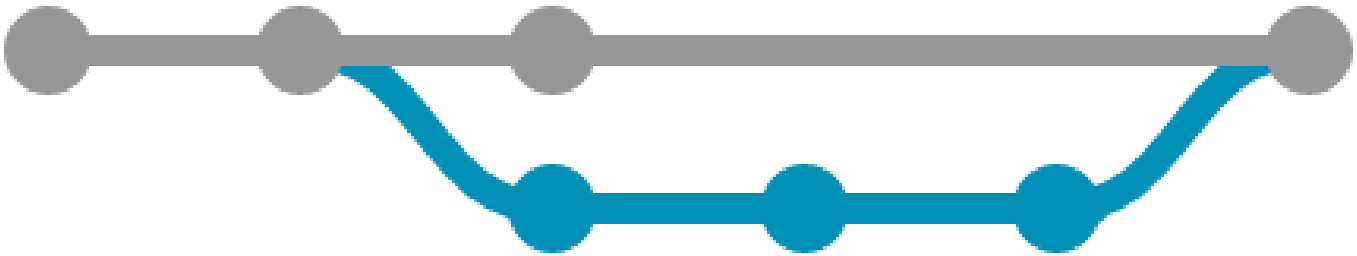

**Figure 4:** Git History is a DAG

# Getting Started with Git

# How to use Git

On windows

- Use cygwin or other terminal

On Mac/Linux

- use your terminal

# Set up your Git Credentials

- Git tracks who did what and when
  - Need to provide some user information

```
$ git config --global user.name "First Last"
$ git config --global user.email "first.last@domain.com"
```

- Use the `--local` option for settings only in one repository.
  - Project folder would need to be a git repository for this to work

# Configure default text editor

- By default Git will open `Vim` if needs you to provide further information:
- Can change to something friendlier:
  - nano:

```
git config --global core.editor "nano -w"
```

  - atom

```
git config --global core.editor "atom --wait"
```

# 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

- Create a text file with some text
- `$ git add [somefile]`
  - add some file to the Git index
- `$ git status`

# What files to keep under Version Control?

- Plain text
- That the computer doesn't generate by running a command
- Not original data sets
- To Version Control:
  - Statistical and Model Code
  - Configuration files
  - Readme files
  - Text of your paper
- Dont Version control
  - pdfs
  - Raw or generated data
  - Word, Excel, Powerpoint files

# 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

- Make changes to an existing file.
- `$ git status`
- `$ git diff`
- - `$ git commit -m "Some message"`

Use short and meaningful messages.

# Simultaneous Adding and Commiting

- `$ 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.
- `$ 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

# 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`

# 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 Commits

- `$ 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.

# Losing Your Head

- If you forget `[yourfile.txt]` in `$ git checkout HEAD [yourfile.txt]`
  - Git message "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

# Ignoring Files in a Git directory

- Can tell Git that some files shouldn't be tracked.
- Specify patterns to be ignored in a file called `.gitignore`, which lives in the project root.
  - Wildcard expressions are your friend here
- Use template `.gitignore` files
  - `$ git add .gitignore -f`
- Is possible to manually specify files to be ignored.

# Branching

# What is branching?

- Branches: different versions of the same files
  - Different Git branches are different a parallel worlds
- Why would you do this?
  - Keep one "good" version & develop code in another
  - Develop different aspects in parallel
  - Can be a way to collaborate concurrently
    - more on this later
- Can integrate git branches easily at future point in time

# `master` is the default branch

- When you create a new repository, typing `$ git init`
  - You start on the `master` branch by default
  - We ignored this terminology at the beginning when explaining output from `git status`
- Rule of Thumb: keep `master` as the current 'good' version

# Git History with Branches

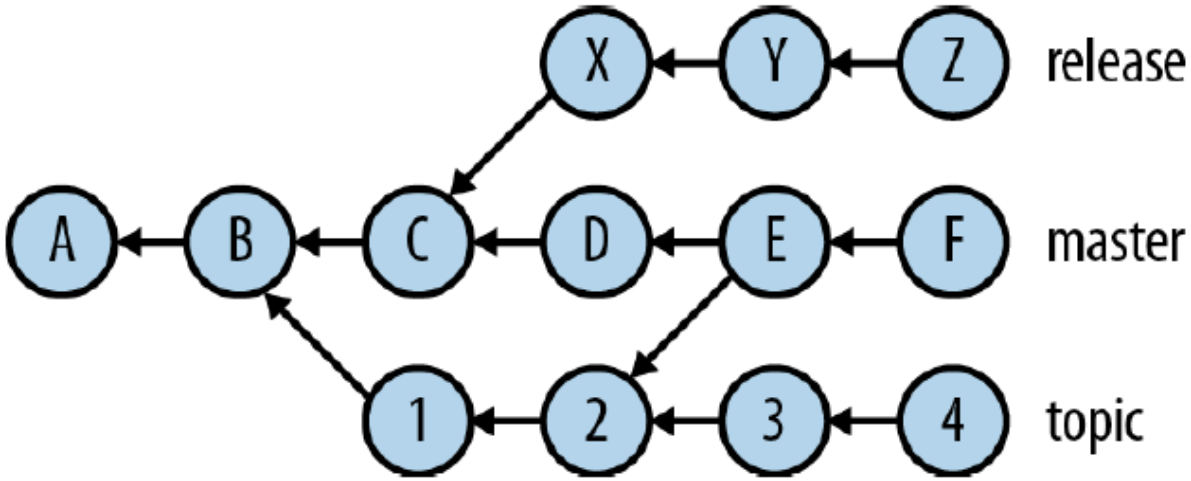


**Figure 5:** Example Git Branches

# 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 [commit-ID]`
  - start a new branch at the commit name `commit-ID` 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

# 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 files
- Make some Commits of your changes
- Move back and forth between your branches

# Merging

# Merging

- Merging: combining the recent changes from several branches into another
- 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]`

# 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

# Useful References

- Books to look up stuff:
  - Loeliger and McCullough (2012)
  - Silverman (2013)
- Stack overflow

# 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
  - Shotts, W.E. (2012). The Linux Command Line. San Francisco: No Starch Press.

# Acknowledgements

This module is based on the 2016 and 2017 versions of the course:

- `Programming Practices For Economists`, by Lachlan Deer, Adrian Etter, Julian Langer & Max Winkler

It is designed after and borrows a lot from:


Suggested Citation:
Lachlan Deer and Julian Langer, 2019, Programming Practices for Research in Economics: Foundations, University of Zurich

# Programming Practices Team

Programming Practices for Research in Economics was created by

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

at the Department of Economics, University of Zurich in 2016.

These slides are from the 2019 Foundations edition, conducted by

- Lachlan Deer
- Julian Langer

# References

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."