TRR 318/RTG

CONSTRUCTING
EXPLAINABILITY

GitLab-Tutorial and Guidelines
for the TRR 318 *Constructing Explainability*

Authors:
Amelie Robrecht
arobrecht@techfak.de,
Vivien Lohmer
vivien.ebben@uni-bielefeld.de

updated version from September 13, 2022

Contents

First we have some regular information on how to read this tutorial and how it was created:
We decided to use Git and GitLab for datastorage, because it offers the opportunity to backtrack changes and recreate previous versions of the files. This tutorial is developed for TRR-members who have never worked with Git or GitLab before and wants to help you through the first steps. We do not want to give a complete documentation for Git and GitLab, but a tool to do the first steps. If you need further information we suggest to use `https://docs.gitlab.com/ee/gitlab-basics/start-using-git.html`.
If you are using Git and GitLab in your TRR318 subproject, we recommend you to have one person in charge. The person does not need to be an expert in GitLab but should be reliable and interested. This GitLab person can always contact us to ask questions and get support.
Finally here are some hints on how to read this Tutorial:

- If no specifications are given always stay with the Default settings!
- If some words in the codeboxes are written in capitals you have to write your personal information (e.g. YOURNAME: here you have to write your name and not YOURNAME)
- the code in the blue boxes has to be tipped into the Terminal/GitBash

But now, let's start.

1 Git vs. GitLab and why you need both

If you have never worked with Git before, it is probably best to have a short look on this two minute Video:
`https://www.youtube.com/watch?v=2ReR1YJrNOM` otherwise this analogy taken from Reddit (thanks to the colleague who passed it to me) might give you a good impression of the idea behind Git and GitLab:

> “ Think of it like this:
> You are working on a paper. You write five paragraphs. Then your friend comes along later, and writes another five paragraphs. Then you edit a few paragraphs, and your friend edits a few more paragraphs.
> At the end of it, there are some issues, but its hard to coordinate who edited what and when.
> With a system like GIT, you would first create the file on the GIT server, and add your changes. Then, you commit those changes to the repository, which will basically say 'Hey, this file exists and looks like this.'
> Then, your friend checks out that repo, pulling the latest changes. He adds some stuff, then commits to the repo. The repo says 'Hey, this file exists and is different from the previous version, so we will save it as version 2.'
> So on and so forth. The GIT repo allows you and your friend to make edits and keep the changes sort of sequential, so if your friend fucks up at version 5, you can say 'GIT, please check out Version 4' and you can get back to fixing things. Each commit has an owner, so you know who did what. And you can compare Version X to Version Y, and see what has changed, making it easier to spot what new changes have been made.
> GIT also allows you to branch. So you and your friend are on version 10 of the doc, but you want to try something new. Your friend wants to continue, so you create a branch called 'MyNewIdea' and start working on that. This is now a separate path of writing, while your friend continues on the original, which is usually considered the master or trunk. Get it? Branches and trunks.
> So GIT basically lets multiple people access a central chunk of what is usually code, and make alterations to it without stepping on each others toes. There's a bunch of other stuff GIT lets you do, but simplified its a way of being able to track changes to a document with multiple users messing with the document, including making branches of the main document for individual development
> GitLab is just a website that works using GIT that has a lot of code. Most of it is opensource, and the idea there is that if you wanted to add some paragraphs from someone on GitLab, you would just check out their GIT repository and add it to your document, more or less. ”

Git is a version control system to locally check changes in your project and push & pull changes from remote repositories like GitLab, it is installed on your system. The changes that you put into Git cannot be seen in GitLab, because they are only stored locally.

GitLab is a service that allows to host a project on a remote repository and have some more features (project management, code sharing, wiki, bug tracking, CI/CD), it allows to work on a project from different systems and with several people. To use GitLab Git is needed.

2 Installation and First Setup

In this section we will install Git and set up the GitLab-Account. Finally we will set up an SSH-key, to simplify your workflow a lot.

2.1 Git Installation

First we have to install Git.
Download Git from `https://git-scm.com/` and install following the instruction for your system.

HINT for MAC:

If you are using MacOS it is the best (but not the only option) to install Homebrew. You can use it at a later state again.

HINT for WINDOWS:

You can either use the *Command Prompt/ Eingabeaufforderung* or *GitBash*, which is automatically installed through the git installation.
I prefer GitBash!

You can check whether the installation worked by checking your current Git version:

Git version

```
git --version
```

2.2 Git Account

Run the following commands in your Terminal (Linux/MacOS) or GitBash (Windows). The commands set or check either your username or your mailing address connected to git. This setup only needs to be done once or if you want to change the deposited information.

You can use whatever name you want to use for git. If the name you use includes a space you have to use quotation marks, I would not recommend to do this.

Username

Changes your Username

```
git config --global user.name YOURNAME
```

Shows your current Username

```
git config --global user.name
```

To change the e-Mail connected to Git please use the Mailadress you will also use to set up GitLab (university mail).
HINT:
Make sure you tipped in email and not mail. You can set a mail as well, but that is a different variable!

Mail

Changes your Mailadress

```
git config --global user.email MAILADRESS
```

shows the currently connected mail

```
git config --global user.email
```

If you want to check all the information that is deposited in Git, you can generate a list:

Summary

shows all configuration information of your Git-Account

```
git config --global --list
```

2.3 GitLab Account

This section will show you how to set up a GitLab-Account.
Please use the account provided by the Bielefeld University (not your private account and not the Uni Paderborn-Account).

If you are working in Bielefeld:
`https://www.uni-bielefeld.de/ub/digital/forschungsdaten/dienste/gitlab/`
HINT:
Use the BITS-Login, not the regular one!

If you are working in Paderborn:
You need to set up a Guest Account. To do this please use your @campus.upb.de or the @uni-paderborn.de address (the registration is only opened for these domains). To do this visit: `https://gitlab.ub.uni-bielefeld.de/users/sign_in` and create an account:

- Click *Register Now*
- Fill in the (Web)-formula and send it
- Now you need to check your mails
- Open the confirmation mail and click on the link

- Now you can login to your Account

Afterwards please send an e-mail to your projects GitLab-person and tell her/him the e-mail you used, so you can be added to the subprojects repository.

2.3.1 SSH key

You do not necessarily have to set up a SSH key, but we would highly recommend it. If you have set up an SSH key for GitLab you do not always have to enter your username and password when pushing files to GitLab, so it helps you to smoothly work with Git and GitLab.

If you want to use SSH, this is how you do it:

1. Type into your Terminal/GitBash

   generate SSH-Key

   ```
   ssh-keygen
   ```

2. stay with the given directory[1]
3. If you already have an SSH-File in that folder it will tell you:

   Overwrite

   ```
   .../.../... already exists. Overwrite (y/n)?
   >> y
   ```

   select *y* to generate a new key!

4. You can enter a passphrase or leave it empty, this is up to you.
5. Make sure to copy the public link, **NOT** the private one.
   You can either copy the public link using the Terminal or by opening it manually (see HINT):

   copy public link

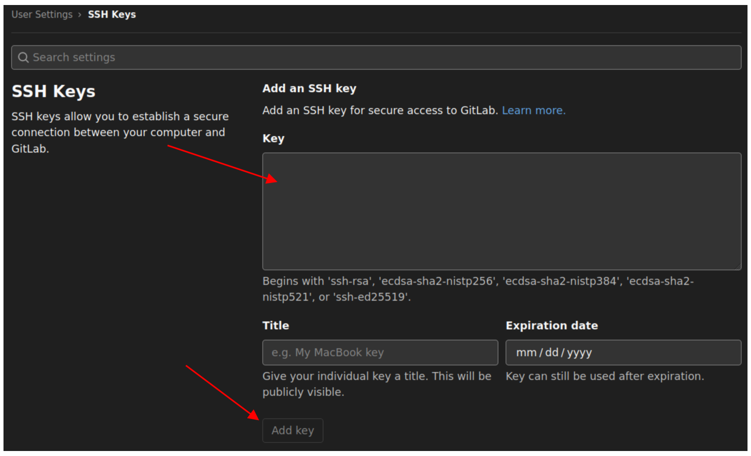
   ```
   cat ~/.ssh/id_rsa.pub
   ```

[1]If you want to save the key in another directory: This video explains how to use cd to change a directory.

HINT:
Open the *id-rsa.pub file* in the *.ssh*-folder with a text editor and copy the content to the clipboard

ATTENTION:
The created *.ssh*-file is a hidden file. Please activate *show hidden files* in the working directory to see the created file (see section 2.3.2.

6. Login to GitLab → Edit Profile → SSH-keys → Paste SSH key from the clipboard (and optionaly name it) and save it (see Fig. 1).



User Settings › SSH Keys

Search settings

SSH Keys

SSH keys allow you to establish a secure connection between your computer and GitLab.

Add an SSH key

Add an SSH key for secure access to GitLab. Learn more.

Key

Begins with 'ssh-rsa', 'ecdsa-sha2-nistp256', 'ecdsa-sha2-nistp384', 'ecdsa-sha2-nistp521', or 'ssh-ed25519'.

Title

e.g. My MacBook key

Give your individual key a title. This will be publicly visible.

Expiration date

mm / dd / yyyy

Key can still be used after expiration.

Add key

Figure 1: SSH keys

2.3.2 Showing hidden files

Windows:

- Select the Start button
- Select Control Panel > Appearance and Personalization.
- Select Folder Options, then select the View tab.
- Under Advanced settings, select Show hidden files, folders, and drives, and then select OK.

Mac:

- Open the folder where you want to search hidden files.
- Press the Command + Shift + . (period) keys at the same time.
- The hidden files will show up as translucent in the folder.

- If you want to obscure the files again, press the same Command + Shift + . (period) combination.

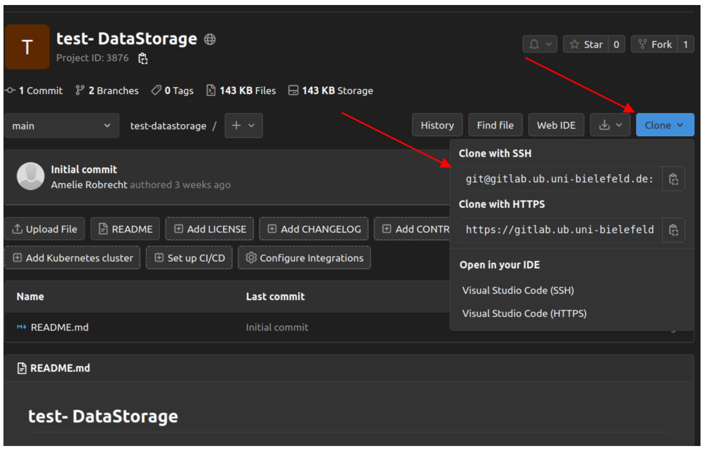
3 First steps

You are done now with setting up Git and GitLab.
Now we will see how to clone an existing repository from GitLab, how to push files after changes were made and how to create branches.

3.1 Cloning an existing repository

Your projects GitLab-person will tell you the link to your projects repository.
HINT:
If you do not have acess to the Repository you probably did not tell her/him your GitLab-Mailadress by now.

1. Clone the repository URL (fig. 2)



```
git@gitlab.ub.uni-bielefeld.de:
https://gitlab.ub.uni-bielefeld
```

Figure 2: clone with SSH

2. change working directory to the folder you want to store the repository in.
   If you never did that before, please watch this short Youtube-Video `https://www.youtube.com/watch?v=1rUFqkRQkok`. It is easier to follow a video in this case.

Path

change the path to the directory you want to save the Repository in

```
cd PATH
```

HINT:

If you are using Windows you can open GitBash in a specific directory by opening the folder and clicking the right mouse button. Then you can select *Git Bash Here* and you do not have to change the directory manually.

3. Clone the repository to Git

Clone URL

clone the URL you copied from GitLab

```
git clone URL
```

Now you have the cloned repository on your system. You can now work on the files, make changes and save them on your system. If the changes are done and you are happy with it, you can first put (add) it toGit and upload (push) the file to GitLab again, so everyone has access to it (section 3.2).

3.2 Pushing new files to GitLab

This short example guides you through your (probably) most frequent workstep: You changed the content of a file and want to put the new version on GitLab now. In this example you added a new word to the file *try.txt*, it was *hello world* before and is *hello new world* now.

Save the file The file should be saved in the cloned gitlab-folder you created in the workstep in section 3.1.

Change Directory

You need to change your working directory to the gitlab-folder as well

```
cd .../TESTPROJECT
```

HINT:

You can use *Git Bash Here* again.

Change the Branch By default you will be in the *master/main* branch of the repository - here you are not allowed to push anything[2]. Depending on where you want to push your data, you need to change the branch. To do

[2]To which branch you can push in GitLab depends on the status you have in the repository. We suggest to only give the GitLab-person the opportunity to push on the main branch, to prevent chaos.

this type in:

Branch

Change to the branch BRANCH

```
git checkout BRANCH
```

To check which branch you are in at the moment use:

```
git branch
```

In our repository we have a branch for each Phd-student. The student assistants do not have their own branch, but push to their superviors branch.

How to create a branch if you need one will be explained in section 3.3

Add file to Git This step adds the file to your local Git, after adding it to Git it is not visible to others and not available in GitLab.

Check Git Status

```
git status
```

This shows the changes you did. It should say: *Changes not staged for commit... modified: try* in red. This means Git noticed the changes in the file but did not do anything about it yet.

Add file to Git

```
git add try.txt
```

Now the file is added to Git.

Check Git Status

```
git status
```

It should say: *Changes to be committed... modified: try* in green. This means you can commit the changes now.

Write a Commit Message A commit message is supposed to sum up the required information about the changes you made. The message should be as short as possible.

Commit message

```
git commit -m "MESSAGE"
```

The message should be short but contain all the necessary information about your changes. Guidelines for commit messages for the Cross-Project-Repository can be found in section 4.

Check Git Status

```
git status
```

It will tell you now: *nothing to commit, working tree clean*. Git accepted the update now, but GitLab did not. To have Git and GitLab at the same level, we have to tell GitLab about the changes as well.

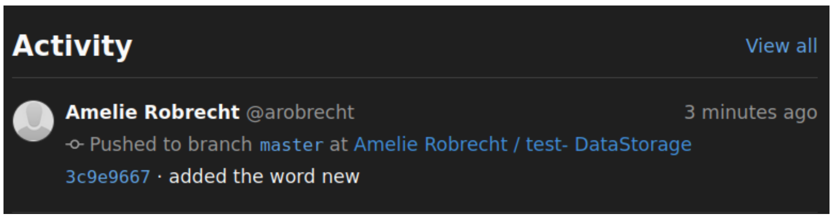
Push to GitLab Now you can push the File from Git to GitLab to make it visible to everyone who has access to the repository. The commit message from the previous step will be displayed as well.

Push to GitLab

```
git push
```

If you added an SSH-key to GitLab it will be pushed straight away, otherwise you need to enter username and password.

Changes in GitLab (fig. 3)



Activity View all

Amelie Robrecht @arobrecht 3 minutes ago
Pushed to branch master at Amelie Robrecht / test- DataStorage
3c9e9667 · added the word new

Figure 3: You can see the changes is GitLab now

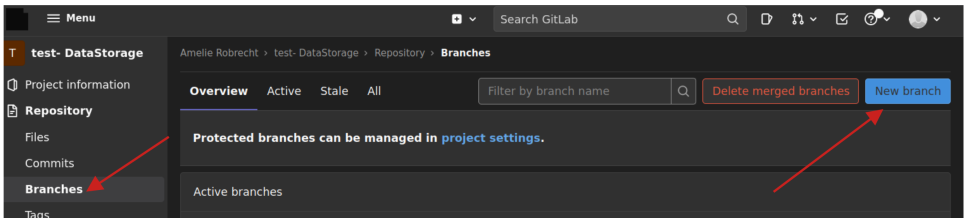
Check Git Status

```
git status
```

Both should be equal now!

3.3 Branches

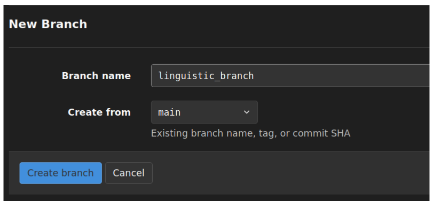
Branches are copies of a repository that exist next to the main repository. It might make sense to start a new branch if you want to make significant changes, work in a completely different directory or try different approaches next to each other. In the end you can merge the branch with the original one to combine them or keep them existing as optional versions.
In our project we are planing to have a branch for each PhD-student[3]. Once in a while the personal branches will be merged to the main-branch initiated either by the GitLab-Person of your subproject or by you sending a merge request (see section 3.4.1.
Create a new branch first:
Go to the GitLab-Repository you want to use, select **Branches** and click on **New Branch** (Fig 4).



Protected branches can be managed in project settings.

Figure 4: select and create new branch

Now you can name your new branch and – if the repository consists of multiple branches already – select the branch you want to copy from the drop down menue (fig. 5):



New Branch

Branch name: linguistic_branch

Create from: main

Existing branch name, tag, or commit SHA

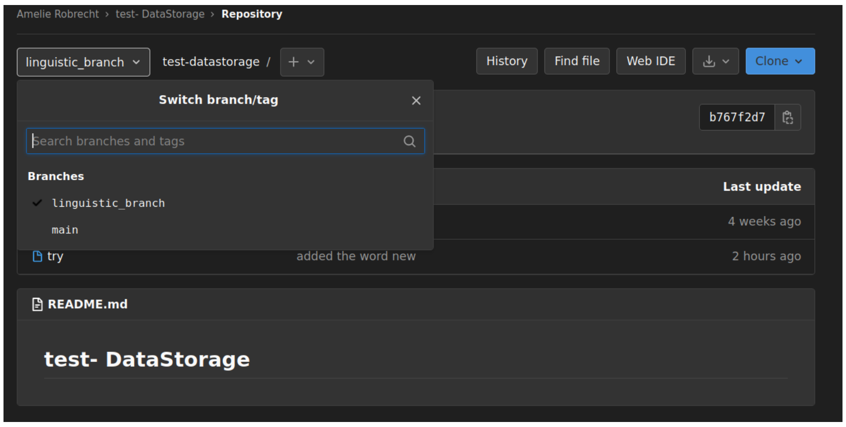
Create branch   Cancel

Figure 5: Type in branch name

Now you can switch between the branches in GitLab, clone them and upload changes to specific branches (fig. 6).

To work with the new branch, just select it in the **Switch branch/tag** drop-down menue and clone it as described in section 3.1.
If you start to work on a new file it is best to create a merge request quite early (you can send a merge request for an empty file as well) so everybody can see, that you are working on this in the main branch.

[3]The research assistants will not have their own branches but will work on the PhD-branch



added the word new

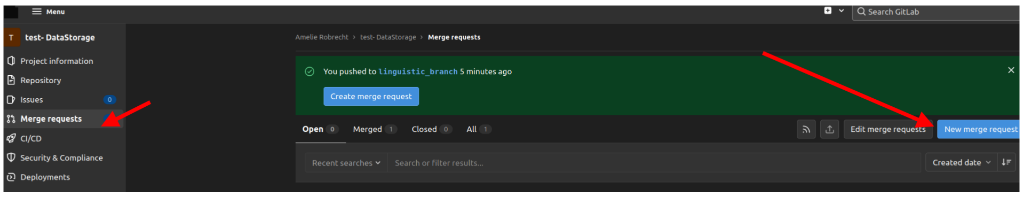
Figure 6: select branch from drop-down

3.4 Merge

In GitLab you can either merge files or branches. We merge branches mainly, so we will focus on that[4].
As all of you will have their own branch to work on and it will be Amelie and Michael to merge everything to the main-branch, you will not be the one answering merge requests. Nevertheless you have to create a merge request as soon as you want your files to be merged to the main branch.

3.4.1 Merging branches

If you made changes in a branch and you want to merge it to another branch you need to click on *Merge Requests* on the left and on *New merge Request* on the right (fig. 7). Now you select the correct branches: The branch on the

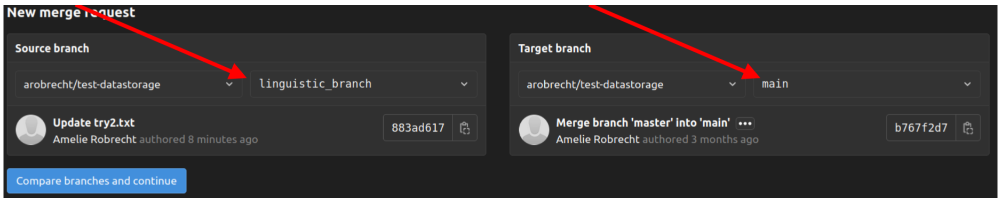


You pushed to linguistic_branch 5 minutes ago

Figure 7: Create a merge request to merge branches

left is the branch that you made the changes on, the branch on the right is that you want to merge with (fig. 8).

[4]If the branches that are merged contain files with the same name, the files are merged into a new file.


New merge request

Source branch

arobrecht/test-datastorage | linguistic_branch

Update try2.txt
Amelie Robrecht authored 8 minutes ago
883ad617

Target branch

arobrecht/test-datastorage | main

Merge branch 'master' into 'main'
Amelie Robrecht authored 3 months ago
b767f2d7

Figure 8: Select the correct branches

Select the Assignee and press *Submit merge request*. Now the Assignee can see the merge request. If there are no merge conflicts the Assignee can just merge the branches.

4 Guidelines for data storage - The fileshare

The following part of the tutorial describes an option, how and where to storage your data. First we give you some general information concerning the fileserver (4.1). Second you will find the scheme to name each file (4.2).

4.1 Where and how to save data: General information

When starting collecting data, one should start of thinking where and how to storage these data as well. Questions that arise could go beyond the basic questions of how much data it will be and where do I store this data? One should think of how can i name each file and folder in a way, that helps me to organize the corpus in a reproducible way as well. Additionally it must be a structure that makes it possible for other persons (colleagues, student assistants,...) to understand this structure and find any kind of data independently. In order to achieve that, you should think about the following things before starting your data collection:

- Which sort of data will be generated?
- Is it only video data?
- Is it a mix of video and audio data?
- Will there be some other documents, which include meta information concerning my corpus?

A good advice might be: Every thing its home.

One possible location to storage your data could be the fileserver with location at Paderborn University. When deciding to storage your data here, please contact Kai Biermeier (*kai.biermeier@uni-paderborn.de*).

To storage the files on the fileserver connect via VPN and go to the following fileshare:

`smb://fs-cifs.uni-paderborn.de/upb/groups/explain-members/data`

IMPORTANT:
Employees from Bielefeld and Paderborn have to dial in via the VPN from their home offices and from Bielefeld University. When you are unsure which VPN Client is the right, please contact the IMT (`https://imt.uni-paderborn.de`).

On this fileserver you can find two folders for your specific poject:

```
Explain_[Project Number]_data

Explain_[Project Number]_documentation
```

How you want to structure your subfolders depends on you. We for instance decided to subdivide Documentation into **Documents** and **Presentations**.

Each folder is for a specific kind of data.

- **Data**:
  Save all kind of (e.g. video- and audio files, transcripts, ...) which are collected during the data set or which are some kind of data, which are necessary for your analysis in `Explain_[Project Number]_data`.
- **Documentation**:
  Save all kind of meta date (e.g. tables, figures, presentations, ...) in `Explain_[Project Number]_documentation`

4.2 Systematic storage scheme

Now we will describe a possible scheme to name each of your files in a way, that helps you and your team to stay organized and find all files as fast as possible.
Before starting with the scheme in detail, we want to give some general rules as suggestions, which might be important if you and your colleagues are using different operation systems (Windows, Mac, Linux, ...). After that the scheme in general will be described, followed by an example. Here we show how two project teams apply the naming scheme.

- Avoid **ä, ö, ü, ß, ...** and all kinds of special symbols like **? !, ...** and so on.
- Avoid **space** in file names.

The scheme works with **units of information** (short: **UoI**). Each individual unit can be understood as a self-contained unit of meaning. The name of each file starts with the **number of trial**, followed by **n units of information** and ends with **number of project**.

Generalized the scheme for each file and folder looks like this:

```
[Number of trial]\_[UoI 1]\_[UoI 2]\_...\_[UoI n]\_[Project Number]
```

Each UoI transports a small amount of information about the file. To ensure that the names of the files remain short, each unit works with abbreviations, which can be chosen for each project and data collection. These abbreviations must be documented.

The following list summarizes the most important points regarding the procedure for naming the files:

- Each name of a file starts with the number of trial, started with **01**.
- Each unit of information[5] (short: UoI) in the name of a file is separated with _ .

[5]A unit of information should include content of the file, e.g. camera perspective, date of trial, or anything else that suits your kind of data.

- If one UoI is not neede, this part is exluded in the name of the file.
- In one unit of information you seperate the words with - .
- Each file is named with the same scheme.

4.3 Example: Application of the scheme in projects A01 and A04

The following chapter shows you an example of the application of the scheme described above in the two projects A01 and A04. Various types of data are collected during each data collection, e.g. *.mp3, .mp4, .mts, .docx, .rtf, ...* . First we will start with all the important **Abbreviations** you need to know for understanding the naming of each file:

EE	Explainee
EX	Explainer
T	Table
VR	Video Recall
KS	Competence Score
C	Camera
A	Audio
VP	Number of trial
P	Phase of experiment

Following the scheme described above(see 4.2), video and audio data of each data set are named as follows:

```
[VP-number]_[perspective]_[phase of experiment]_[project number]
```

As shown in the following table each unit is defined as a self-contained unit of meaning. For example **[perspective]** describes who is shown or can be heard in the file.

[VP-number]	[perspective]	[phase]	[projectnumber]
VPXX	C-EE	P0	A01
	C-EX	P1	A04
	C-T	P2	A01-A04
	A-EE	P3	
	A-EX	P4	
	VR-P2	P5	
	VR-P4	P6	
	KS-EE		
	KS-EX		

Table 1: Possible items to fill in each bracket

For better understanding of the scheme and practice, we conclude by giving a few examples of different file types.

Example 1: A video file, which shows EX, collected during the first recording by A01. The file includes the second phase of the experiment:

```
VP01_C-EX_P2_A01
```

Example 2: A project file to synchronize video and audio from the first recording from A01:

```
VP01_A01
```

Example 3: A transcript of the third recording in project A04 of the third phase of experiment.

```
VP03_P3_A04
```

Thank you for reading this short tutorial on Git an GitLab. We hope it was helpfull to you. In addition to this tutorial we created a Git/GitLab Cheat-sheet, which sums up all the needed commands for the pulling/pushing workflow. You are welcome to use it while working with Git and GitLab.
If you have any comments, questions or additional ideas on this tutorial, please feel free to contact us.

Amelie Robrecht
subproject A01
Social Cognitive Systems Group
Bielefeld University
arobrecht@techfak.de

Vivien Lohmer
subproject A04
German Linguistics/Didactics of Language
Bielefeld University
vivien.ebben@uni-bielefeld.de