TIB LEIBNIZ-INFORMATIONSZENTRUM TECHNIK UND NATURWISSENSCHAFTEN UNIVERSITÄTSBIBLIOTHEK

Leibniz Universität Hannover

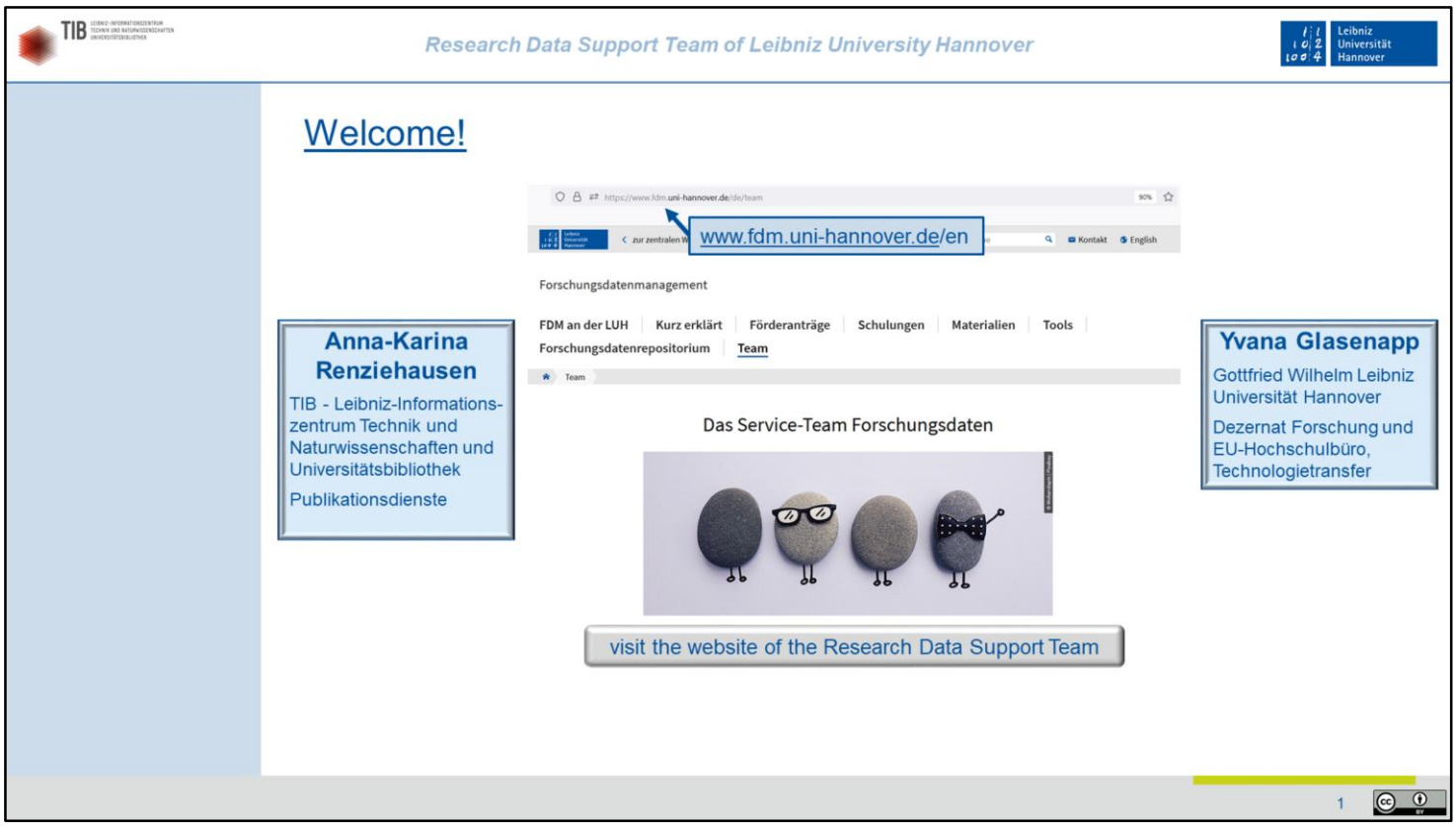
Love Data Week 2024

Herzlich Willkommen!

Software Management- How to handle research software

12.02.2024, 14 Uhr

Service-Team Forschungsdaten | Love Data Week 2024 | #LoveData24



We are Anna Renziehausen from the TIB Publishing Services and Yvana Glasenapp from the LUH Research and Transfer Services, and we will guide you through this course. We both belong to the Research Data Support Team, which also comprises other colleagues from our departments and from the Leibniz University IT Services, known as LUIS. If you want to know more about our training courses, counselling and support, have a look at our website, where you will find extensive information on research data management in general. We are also happy to provide individual counsel to LUH members.

Visit the website of the Research Data Support Team

Content of this course

- Why is data management important?
- How to make research software re-usable
- How to publish research software and code
- Supporting services and initiatives

In this course, we would like to give you an introductory overview of the most important aspects of dealing with research software.

Why is data management important?

- ✓ you keep an overview
- ✓ team work is easier
- ✓ you can safeguard high quality standards in research
- ✓ you save time and avoid stress
- ✓ you comply with official requirements
- ✓ Data management is the basis for Open Science
- ✓ You gain recognition for published data and software

further reading

The Thuringian Competence Network for Research Data Management compiled some "Research Data Scarytales". These are true stories about data management failures and their consequences.

go to the „Scarytales"

The opinion is still widespread that a planned and systematic data management consumes valuable time and resources that should be spend preferably for the actual research work. All too often, a doubled or tripled amount of the supposedly saved-up resources have to be spent afterwards in order to iron out the negative consequences of a negligent data management, at least partially.

If you manage your data well right from the start, there are many advantages:

- You maintain an overview because you can reliably find data and information again.
- You work together much more effectively as a team because you have agreed on common standards and procedures.
- The quality of your data and the research results derived from it is assured because you have established meaningful review mechanisms.
- You save an enormous amount of time and energy, especially at the end of a project because your data is already prepared and documented. You have everything at your fingertips and can even reconstruct what you did at the very beginning.
- You comply with various formal requirements, such as those defined in laws, guidelines or funding conditions.
- Engaging in Open Science practices is encouraged by various institutions like the EU or the UNESCO and is gaining more and more importance.
- There is also a shift towards a broader and more diverse recognition of research outputs beyond classical text publications. One example is the new CV template by the DFG, in which you can now add up to ten published contributions that are not scientific articles, e.g. data publications, software, blog contributions and contributions to infrastructure and science communication.

Take a look at our reading tip to see what can go wrong without proper data management:

Research Data Scarytales

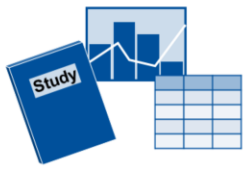
How to create re-usable research software

- Documentation of software projects
- The FAIR principles for research software
- Developing a software management plan (SMP)
- Foundation: How to set up a project
- Best practices of software projects
  - Tests and Test-driven development (TDD)
  - Refactoring
  - Clean code
  - Error messages
  - README

In this chapter, we will show concepts, methods and tools you can use to create research software that is reusable by others and can be maintained over a long time.

Documentation of software projects

Data



But most researchers

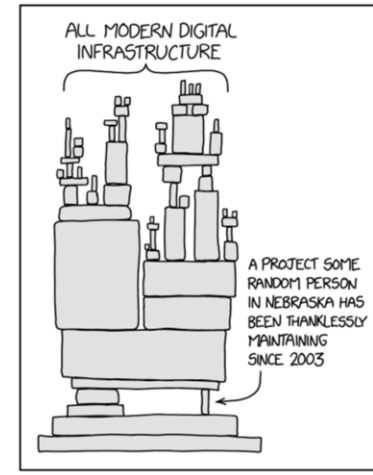
is not independent from

≠

Research software

Are no software developers

"Data without software are just numbers" (Davenport & Grant, 2020)



https://xkcd.com/2347

In many projects, software is seen as a tool to analyse the data in a certain way to create analysed data as research outputs. But software and the underlying code is a type of research data, too! The way software projects need to be documented to become FAIR software are similar to handling other types of data, but there are some differences. Nicely though, the way software projects are build up offers many points where RDM practices can be included quite easily.

Read the Essay: "Data Without Software Are Just Numbers"

The FAIR Principles for research software

F: Findable

This means that the software and all important related information can be easily found by both humans and machines.

F1. The software, individual important components of the software and the different versions of the software all receive persistent and unique identifiers (PID), e.g. DOI.

F2. The software is described in detail with metadata, which means that all important information for use and subsequent use is included.

F3. Metadata clearly and explicitly include the PID of the software they describe.

F4. Metadata is also FAIR, searchable and can be found and used by metasearch engines.

✓ Use persistent identifiers and rich metadata

Findable

RDA RESEARCH DATA ALLIANCE

The important thing in documenting software projects is planning the steps you want to implement during code development, and then be consistent with this practice. Having created the code of your research software in a well documented way will not only help you enjoy using the software in the future, it is also crucial to manage und reuse data which was processed with the software.

You can use the FAIR Principles for Research Software as an orientation which aspects you need to consider to build well-documented and reusable software. In this and the following slides we have briefly lined out what the principles contain, but we recommend reading the full article:

FAIR Principles for Research Software (FAIR4RS Principles)

Source of the graphical elements illustrating the four FAIR principles:

National Library of Medicine

The FAIR Principles for research software

A: Accessible

Which means that the software and its metadata is retrievable via standardized protocols

A1. For this, the protocol should fulfil some requirements, it should be open and free as well as universally implementable. It should also enable an authentication and authorisation procedure, if necessary.

A2. In addition the metadata are accessible, even when the Software is no longer available.

✓ Make software and metadata retrievable

Accessible

Accessibility can be achieved, for example, by enabling the software to be downloaded via the browser using https (A1) and the metadata can be accessed independently of the software, even if the software is no longer accessible (A2).

The FAIR Principles for research software

I: Interoperable

For software, being interoperable means the possibility to exchange data between independent software. (It differs from data in the way that independent software can not be combined like data to form a new data set.)

I1. When software interacts to exchange data, the exchange protocol need to be clearly described. Domain-relevant standards should be used and APIs should be documented in human and machine readable form.

I2. Software should include references to external data, software or objects, which are required to execute the software. Example: software X is implemented using software A (a programming language)

✓ Use APIs, standards and references

Interoperable

To incorporate a new part of software into the data analysis, it might be necessary to exchange data between different software. To make this possible, exchange protocols should be well described and domain-specific standards should be followed. In the software documentation, you should reference all other, external element that are needed to execute the software, so others do not have to search for those.

The FAIR Principles for research software

R: Reusable

This means that you can not only use the software as it is, but have further information to understand, modify and built new versions upon this software.

R1. The software is described with relevant attributes.

R1.1. A license is given and should be as unrestrictive as possible.

R1.2. Software is associated with detailed provenance: This is a metadata which includes the history of the software, how it came to be and who contributed to it.

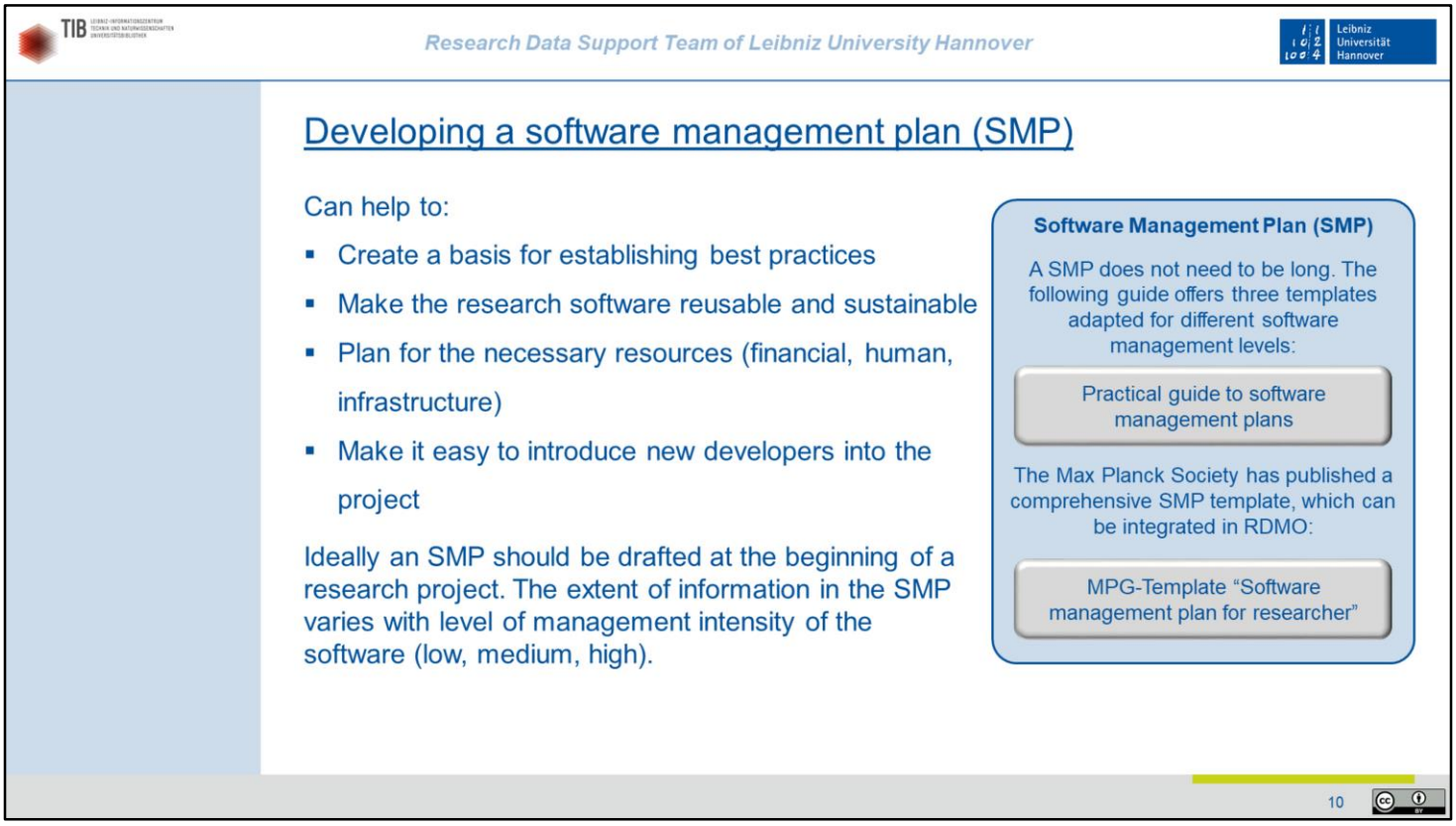
R2. Include references to other software which is necessary to compile and run the software.

R3. Your software should meet domain-relevant community standards and coding practices. This can be the choice of programming language, testing methods or file formats.

✓ Document, licence, and follow community standards

Reusable

If you simply publish your software with limited documentation, other people might be able to use it as it is. But to get a real insight into the structure of the software and to be able to modify it, some more steps need to be undertaken. To make clear what others can or can not do with the software, add a licence (R1.1). If possible, make it unrestrictive to further uses. The provenance of the software (R1.2.) can be really helpful to show people that the code comes from a trustworthy source. If your software relies on other software packages, these need to be named (R2.). You will make reuse of your software more accessible to others, if you use coding practices that are familiar to others. Already when planning your project, check if there is a reason to deviate from standards and broadly accepted practices (R3.).



It is always a good idea to gather your thoughts when you start a new software project. A SMP helps you to do this in a structured way. The resulting document can be used by yourself and others to create, maintain and reuse the software in the future.

In the next two slides we will look at the possible contents of an SMP. It should be noted that software comes in many different forms: from individual scripts to complete frameworks. These require different levels of management. Depending on this, one should decide which of the points make sense to include in one's own SMP and which do not.

We also have two worthwhile reading tips:

Practical guide to software management plans

MPG-Template "Software management plan for researcher"

Developing a software management plan (SMP) part I

- Purpose
- Version Control
- Repository
- User Documentation
- Software licensing and compatibility
- Deployment documentation

Figure 1. Software Management Plan requirements grouped by their focus.

Graphic from: Martinez-Ortiz, C. et al. (2022). Practical guide to Software Management Plans, p. 17. https://doi.org/10.5281/zenodo.7185371

An SMP makes explicit what research software does, who it is for, what the outputs are, who is responsible for the release and to ensure that the software stays available to the community (and for how long).

Here we have now listed some points that could be addressed in an SMP. In the following, we would like to give a few examples of the questions that should be answered under each aspect:

- Purpose: Briefly describe the software, its purpose and the target group.
- Version control: How do you version the software? E.g. with Git?
- Repository: Will you publish your software? If yes, how and where? If no, why not?
- User documentation: How do you document your software for users? Where can it be found (link)? How will you document your software's contribution guidelines and governance structure
- Software licensing and compatibility: Under which license will you release your software? Have you ensured that the licence you have chosen is compatible with the licences of any downstream elements (e.g. libraries)?
- Deployment documentation: How and where will the installation requirements be documented (link)?

Developing a software management plan (SMP) part II

- Citation
- Developer documentation
- Testing
- Software Engineering quality
- Packaging
- Maintenance
- Support
- Risk analysis

6.1.4. Summary of SMP templates developed for three management levels

Core requirement (Section 5.1)	Software management level (Section 6.1)		
	Management level: Low (6.1.1)	Management level: Medium (6.1.2)	Management level: High (6.1.3)
Purpose	×	×	×
Version control	×	×	×
Repository		×	×
User documentation		×	×
Software licencing and compatibility		×	×
Deployment documentation		×	×
Citation		×	×
Developer documentation		×	×
Testing		×	×
Software Engineering quality		×	×
Packaging		×	×
Maintenance		×	×
Support			×
Risk analysis			×

Table 4. Core requirements of an SMP for software grouped by management level.

Table from: Martinez-Ortiz, C. et al. (2022). Practical guide to Software Management Plans, p.29. https://doi.org/10.5281/zenodo.7185371

- Citation: How should users cite your software? A CFF file is particularly suitable for this (Citation File Format)
- Developer documentation: How is your software documented for future developers?
- Testing: How is the software tested and where are the test results published (link)?
- Software engineering: Do you follow any standards or guidelines to ensure software quality? If yes, which ones?
- Packaging: How will your software be packaged and distributed?
- Maintenance: What level of support will be provided for users of the software and how will this support be organised?
- Support: How will support be ensured in the long term?
- Risk analysis: Describe the main external factors that should be considered by developers and users of the software. E.g. data protection or information security.

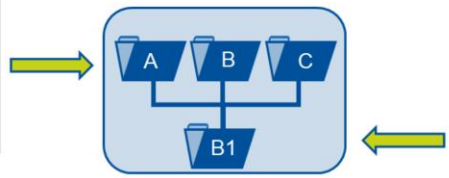
As you can see, such an SMP can be quite extensive, but this largely depends on the nature of your software project: if it is a script, of course, you do not have to provide information on all points. If it is a large software project, more aspects need to be considered. The table on this slide gives a good overview of this.

Small software projects typically require a low level of management, while large projects involve a high level of management.

Foundation: How to set up a project

Create a similar folder layout:

- Licence
- Readme
- Experiments (chronological)
- Provenance
- Scripts



Logical order in subfolders:

- Main script
- Data-generating scripts
- Raw data
- Post-processed data

Modular naming scheme:

Example: 230701_PuppyCuteness_Treats_Cheese_Camembert_v03.fff

Data format: Better not choose "Fancy File Format"

Find a file format

There are some things you can do right from the start when you begin a new software project. These will help you to keep the overview of your work and will make it easier to gather everything you need for software publication.

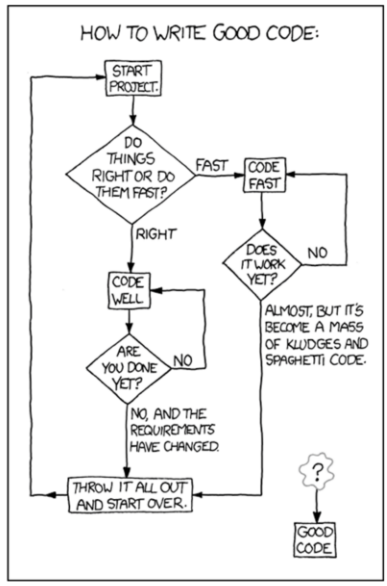
Especially when you are going to use the same scripts for different data, it is helpful to think of a folder layout which contains all elements you need. Now you can easily navigate in the individual project folders, as they all follow the same structure. The same can be done for sub-folders of experiments.

The names of files should also follow a logical scheme that contains all necessary information to identify files.

As long as it is only you who is working with the data, you might use any file format you like or you feel suits your need best. But as soon as you make data public, please do not forget to convert your data into some format that is non-proprietary, can be read by many programming languages and ideally is widely used in your community.

Interactive board of common file formats

Best practices of software projects



There are a number of published „Best practice“ guides for software development in different scientific fields. We have collected some general practices that help you to write FAIR software.

Ten simple rules for documenting scientific software

Oliver Melchert's best practices for small computational projects

Suresoft
SUSTAINABLE RESEARCH SOFTWARE

What can you do to write good, well documented code? Luckily, many people and initiatives share their experiences, which you can adapt in your daily work:

Ten simple rules for documenting scientific software by Benjamin Lee

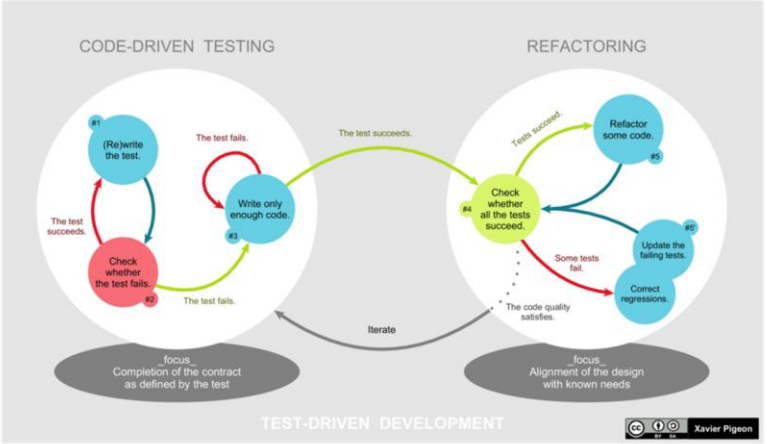
RDM in practice: Best practices for (small) computational projects by Oliver Melchert, LUH

Suresoft: Sustainable Research Software project by TU Braunschweig and FAU Erlangen-Nürnberg

In the following slides, we will present some general best practices which are not specific to a research field. Step by step, you can start to implement these practices in your daily work – it will help you and others to continue working on your software projects in the future!

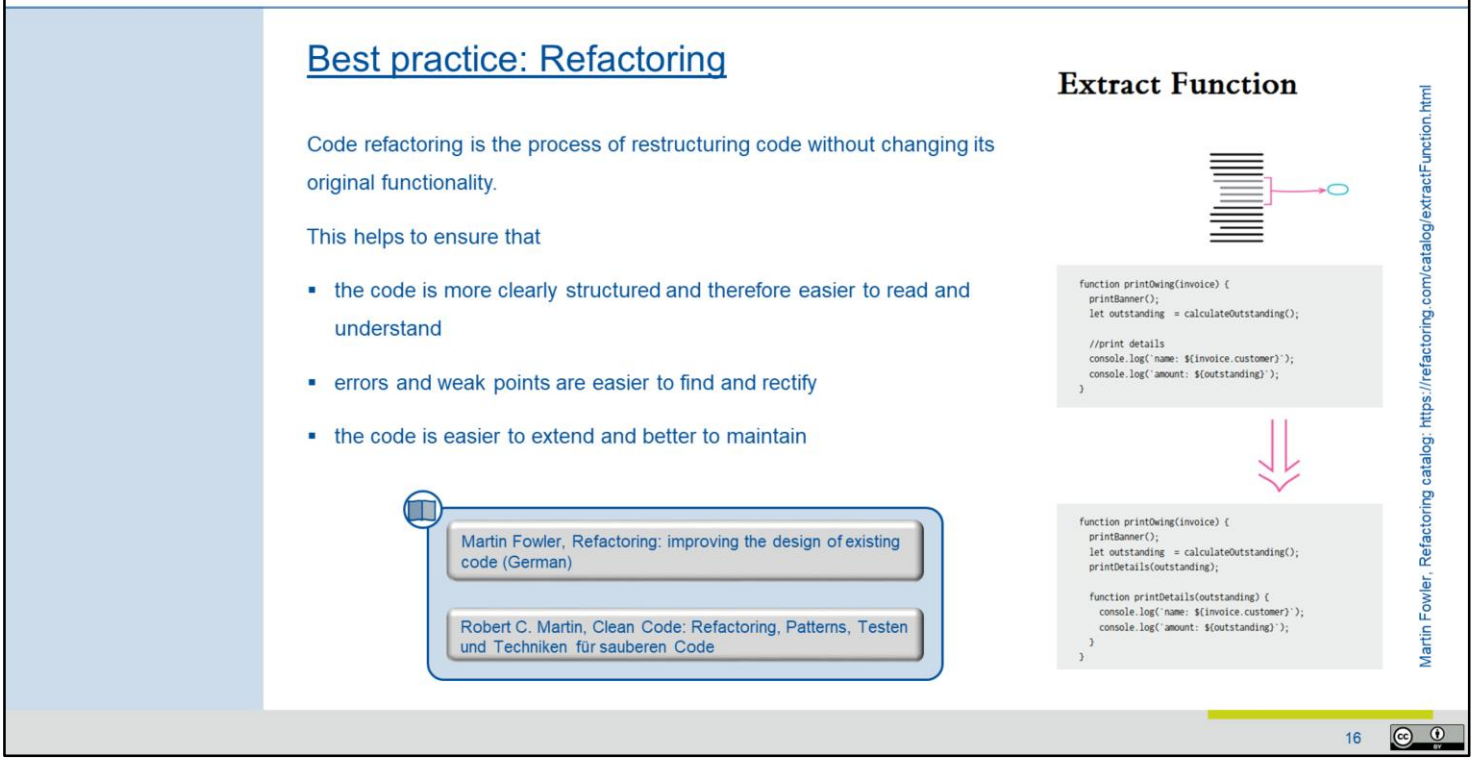
Best practice: Tests and Test-driven development (TDD)

- Include testing in the development process
- Each new piece of code is tested for functionality before it is added to the main branch
- Best practice: add incremental changes!
- TDD: First create a test for the desired functionality, then write the code that is able to fulfill the test requirements



Testing is a useful way to detect errors in your code right away. There are different approaches and methods of testing, maybe one of them is already established as practice in your working group. If not, you should consider to introduce testing as a regular habit. One way is to test any newly written parts of your code before adding it to the main branch. You can correct any problems that the new part might cause right away.

A more strategic approach is called test-driven development (TDD). Here, you will first write a test for the new requirement you want to add to your software. Then, only enough code to fulfil this requirement and to pass the test is added. This method is closely connected to refactoring – see next slide!



One method that we would like to mention is "refactoring". Code refactoring is the process of improving the quality of software code by changing some of its parts, deduplicating the code base, eliminating unnecessary dependencies without changing the external behaviour of the program.

Refactoring is a more time-consuming process, so you should plan it specifically into your workflow, but it's worth it! Here you can see some examples: refactoring.com

In addition to the points mentioned on the slide, it would also be a good reason to start refactoring if you observe logical repetitions or circular code structures, if problems occur with a certain part of the code or if the debugging process takes longer than expected. Here are a few tips on refactoring:

- Refactor before you add new functions or updates to existing code
- Refactor in small steps and regularly
- Troubleshooting and debugging should be done separately

Important refactoring practices are:

- Use meaningful names
- Separate responsibilities and modularise code
- Strengthen cohesion: Ideally, a code component has only one well-defined task
- Reduce coupling: Reduce the degree of dependencies between code components

In our reading tip you will find literature with further helpful tips on refactoring.

Best practice: Clean code

- There are different options how to write names, comments, functions and classes in your code
- Even though a “messy” written code might work, it is not re-usable
- Changing small habits can make a big difference!

In this example, it is better to define a function which contains the for-loop:

Do not

```
input_number = int(input("Enter a number"))

# check if input_number is a prime number
is_prime = True
for i in range(2, input_number):
    if input_number % i == 0:
        is_prime = False
        break

print(input_number, "is a prime:", is_prime)
```

Do

```
def is_prime(number):
    prime_flag = True

    for i in range(2, number):
        if number % i == 0:
            prime_flag = False
            break

    return prime_flag


input_number = int(input("Enter a number"))
print(input_number, "is a prime:", is_prime(input_number))
```

https://suresoft.dev

Your code is considered “clean” if it is easy to read and understand, and therefore easy to maintain and modify. There are some practice with regards to the use of names, comments, functions and classes which make the difference between messy and clean code.

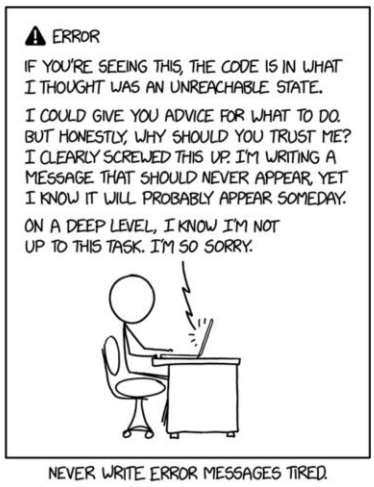
When we look at the use of comments, it is important to see them as part of your software documentation. They help you remember your own thought processes and considerations when you are already working on the next project, and help other people read and understand your source code. Comments are also valuable when troubleshooting. Nevertheless, you should find a middle ground: as many comments as necessary, as few comments as possible. Comments do not replace well-written code!

We recommend to have a closer look at the examples shown in the “Clean Code” course by Suresoft:

Suresoft Knowledge Hub: Clean Code

Best Practice: Error messages

- Write error messages
- Provide solutions or where to find the information relevant to fixing the error.



https://xkcd.com/2200

Graphic: Unknown, via: https://community.spiceworks.com/topic/2105369-10-hilarious-error-messages-facepalm-worthy-computer-prompts-that-make-no-sense

A good error message should state what the error is, what state the software was in when the error occurred, and how to fix the error or where to find the information relevant to fixing the error.

Best practice: README Part I

- Give an overview of the project
- List the names and affiliation of people involved
- Environment and dependencies needed to run the code

Environment and dependencies

The provided code is written using Python (2.7.15+) and requires the functionality of

- numpy (>=1.8.0rc1)
- scipy (>=0.13.0b1)
- matplotlib (>=1.2.1)

All-optical Supercontinuum Switching (code and data)

Code repository for the article

"All-optical Supercontinuum Switching"

Oliver Melchert (1,2,3), Carsten Brée (4), Ayhan Tajalli (2), Alexander Pape (2,5), Rostislav Arkhipov (6), Stephanie Willms (1,2), Ihar Babushkin (1,2), Dmitry Skryabin (7), Günter Steinmeyer (8,9), Uwe Morgner (1,2,3), and Ayhan Demircan (1,2,3)

1. Cluster of Excellence PhoenixD, Welfengarten 1, 30167, Hannover, Germany
2. Institute of Quantum Optics, Leibniz University Hannover, Welfengarten 1 30167, Hannover, Germany
3. Hannover Centre for Optical Technologies, Nienburgerstr. 17, 30167, Hannover, Germany
4. Weierstraß Institute for Applied Analysis and Stochastics, Mohrenstraße 39, 10117 Berlin, Germany
5. VALO Innovations GmbH, Hollerithallee 17, 30419, Hannover, Germany
6. St. Petersburg State University, Universitetskaya nab. 7/9, St. Petersburg 199034, Russia
7. Department of Physics, University of Bath, Bath, BA2 7AY, UK
8. Max-Born-Institute (MBI), Max-Born-Str. 2a, 12489 Berlin
9. Institut für Physik, Humboldt-Universität zu Berlin, Newtonstraße 15, 12489 Berlin, Germany

This repository contains code and data analysis scripts for reproduction of simulated data and a draft versions of Figures 3 of the article.

The provided code implements the nonlinear propagation of optical pulses in a NLPM750 photonic crystal fiber in terms of the generalized nonlinear Schrödinger equation, including the effets of dispersion, self-phase modulation, self-steepening, Raman effect, and quantum noise.

Most likely, the README file is the only documentation other users will read, so the README file should include instructions on how to install and configure the software, where to find the full documentation, under what licence it is released, how to test it to ensure functionality, and your acknowledgements.

Best practice: README Part II

- Details about included material
- License, Citation & Acknowledgements

Subfolder `/scripts` contains:

- `pyGNLSE.py` : driver script implementing the genaralized nonlinear Schrödinger equation.

License

This project is licensed under the MIT License - see the LICENSE file for details.

Acknowledgments

This work received funding from the Deutsche Forschungsgemeinschaft (DFG) under Germany's Excellence Strategy within the Cluster of Excellence PhoenixD (Photonics, Optics, and Engineering – Innovation Across Disciplines) (EXC 2122, projectID 390833453).

Included materials

The repository follows a modular structure:

```
numExp01_FIG03_propagationDynamics
├── data
├── FIG03_subfigures
│   ├── FIGS
│   │   ├── GNLSE_BlowWood_tMax8000.000000_Nt32768_zMax600000.000000_Nz15
│   │   ├── GNLSE_BlowWood_tMax8000.000000_Nt32768_zMax600000.000000_Nz15
│   │   └── GNLSE_BlowWood_tMax8000.000000_Nt32768_zMax600000.000000_Nz15
│   ├── getFigures.sh
│   ├── main_FIG03_subfigure.py
│   └── spectrogram.py
└── main_NLPM750_propagationDynamics_qNoise.py

provenance_data
├── dataProvenance.sh
├── get_info.py
└── provenance_numExp01.log

scripts
└── pyGNLSE.py

src
├── data_handler.py
├── raman_response.py
├── solver.py
└── utils.py

LICENSE
Readme.md
```


Most likely, the README file is the only documentation your users will read, so the README file should include instructions on how to install and configure the software, where to find the full documentation, under what licence it is released, citation requirements, how to test it to ensure functionality, and your acknowledgements.

How to publish research software and code

- Why publish your data and software?
- Preparation of your software publication
  - Version control your documentation
  - Use automated documentation tools
  - Software citation
  - User guide and examples
- Publishing software
- Licences
- Linking journal articles with related data and software

In line with good research practice, you should make your scientific results, including research software, openly available. In this chapter we will show you where you can publish code and software and how licences work.

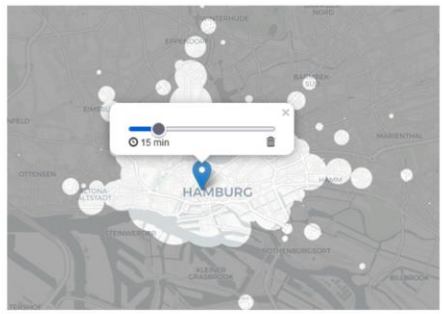
Why publish your data and software?

Open data are valuable and can be used to create new research output or tools of public interest:

- Participation: findtoilet.dk
- Empowerment of citizens: mapnificient
- Innovation: EHEC outbreak genome analysis

Reasons to publish your research software:

- To enable software citation
- To make the software FAIR
- To let software count towards evaluation



Mapnificent.net

Making your data and software ready for publication might include some effort and workload. But there are various reasons why it is a good idea to publish your data and software: We have collected some examples of tools and innovations which are build upon openly available data.

Findtoilet

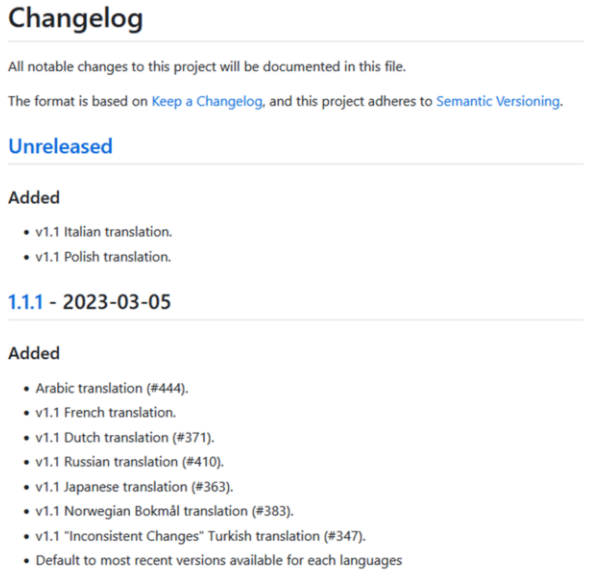
Mapnificent

GitHub E. coli O104:H4 Genome Analysis Crowdsourcing

In the academic field, fulfilling the requirements of funding agencies is often the main motivation to publish data and code. The importance of access to and reuse of research software is getting more and more attention. So by making your software public, it can be cited and linked with other research outputs. Fulfilling the FAIR principles will make the software attractive for others to use or reuse. Software as research output is also getting more value in the academic rewards and recognition system, so you can include published software in your list of research outputs.

Software publication: Version control your documentation

- Make it clear which documentation applies to which software version
- A changelog can make this task much easier
- There are useful services that can help you do this.



Changelog

All notable changes to this project will be documented in this file.

The format is based on Keep a Changelog, and this project adheres to Semantic Versioning.

Unreleased

Added

- v1.1 Italian translation.
- v1.1 Polish translation.

1.1.1 - 2023-03-05

Added

- Arabic translation (#444).
- v1.1 French translation.
- v1.1 Dutch translation (#371).
- v1.1 Russian translation (#410).
- v1.1 Japanese translation (#363).
- v1.1 Norwegian Bokmål translation (#383).
- v1.1 "Inconsistent Changes" Turkish translation (#347).
- Default to most recent versions available for each languages

Screenshot: https://github.com/olivierlacan/keep-a-changelog/blob/main/CHANGELOG.md

Even a small change in the default settings in a new software version, hidden from the user, can have a big impact on the results. So it is very important to version your documentation and make it clear which documentation applies to which software version. That is, keep older versions accessible to users. Keeping a changelog in your documentation can make this task much easier.

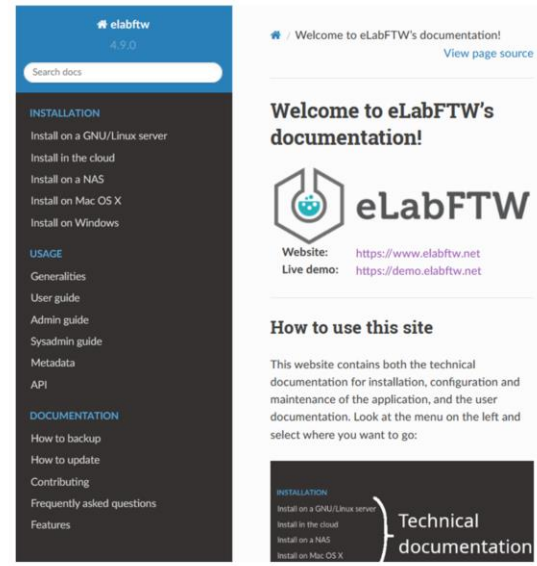
Useful services, which will archive every version of your documentation, are: readthedocs and zenodo.org

Best Practice: Use documentation tools

- Documentation of a project is the alpha and omega
- Complete and prompt documentation is essential
- Use tools to create a nice documentation site which helps others to use your software

Sphinx Python Documentation Generator:

```
docs
├── build
├── make.bat
├── Makefile
└── source
    ├── conf.py
    ├── index.rst
    ├── _static
    └── _templates
```



The documentation of a project is the alpha and omega. Without documentation, decisions and results cannot be traced, reproduced and reused. Documentation should always be done in a timely manner so that nothing important is forgotten. Nevertheless, one can try to simplify this important work step.

For example, there is software that can read your comments and use them to create detailed documentation, like Sphinx (sphinx-doc.org) or Doxygen (doxygen.nl).

Read the docs (readthedocs.com) for example, is a documentation hosting platform that can help you version your documentation.

Software publication: Software Citation

- Provide information on how to cite your code
- Be sure to always properly cite other people's code that you reuse as well

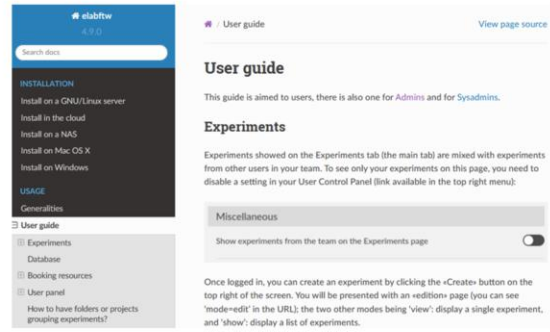
```
# This CITATION.cff file was generated with cffinit.
# Visit https://bit.ly/cffinit to generate yours today!

cff-version: 1.2.0
title: Cheese_TAX_Alert
message: >-
  If you use this software, please cite it using the
  metadata from this file.
type: software
authors:
  - given-names: Sheryl
    family-names: Mc Sniff
    email: SMS@gmail.com
    affiliation: University of Dogford
repository-code: 'https://git.eu/McSniff/cheesetaxalert'
repository: 'https://doi.org/10.123456789/mcsniff'
abstract: >-
  the software can be installed in a refrigerator so that as
  soon as the refrigerator door is opened, the Puppy
  receives a warning signal that the Cheese Tax must be
  collected.
keywords:
  - puppy
  - cheese
  - tax
license: MIT
version: '1.0'
date-released: '2023-09-01'
```

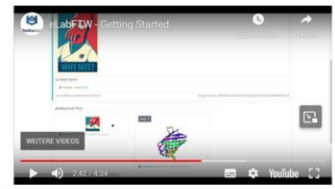

Ideally, include the PID (e.g. DOI), a BibTeX entry and a written reference to your publication in your README file and provide a "CITATION" file in CFF format, so that other researchers who want to cite your code have all the important details. You can use the following tool to create a CFF file: citation-file-format.github.io

Software publication: User guide and examples

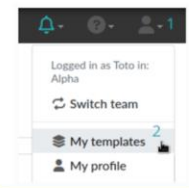
- People might not use your software if it takes too much time to get started
- Add examples, a tutorial, videos – anything that helps to show the functionalities of your software
- If you have numerous examples, use a special section or directory
- Try your user guide on someone who does not know your software yet



Video tutorials



Screenshots



Bring examples that show the main functionality of your software. The more examples, the better. If the examples go beyond the scope of the documentation, they can be moved to a separate directory. Just make sure it is easy to find.

Publishing software

- Publish software in a publication repository (like Zenodo or your institutional data repository)
  - This gives you:
    - An unique and persistent identifier (e.g. DOI)
    - An identifier that points to the specific version of your software
- Provide citation metadata for your software (via CFF)
- License your code
- Possibility of additional publication in a software journal

Repository publication → software journal → papers

Important: The source code repository (e.g. GitHub, GitLab) is not a publication repository!

What should be considered when publishing?

First, that you get a unique and persistent identifier, for example a DOI.

Therefore, you should publish your research software in a publication repository, such as a subject-specific or institutional repository.

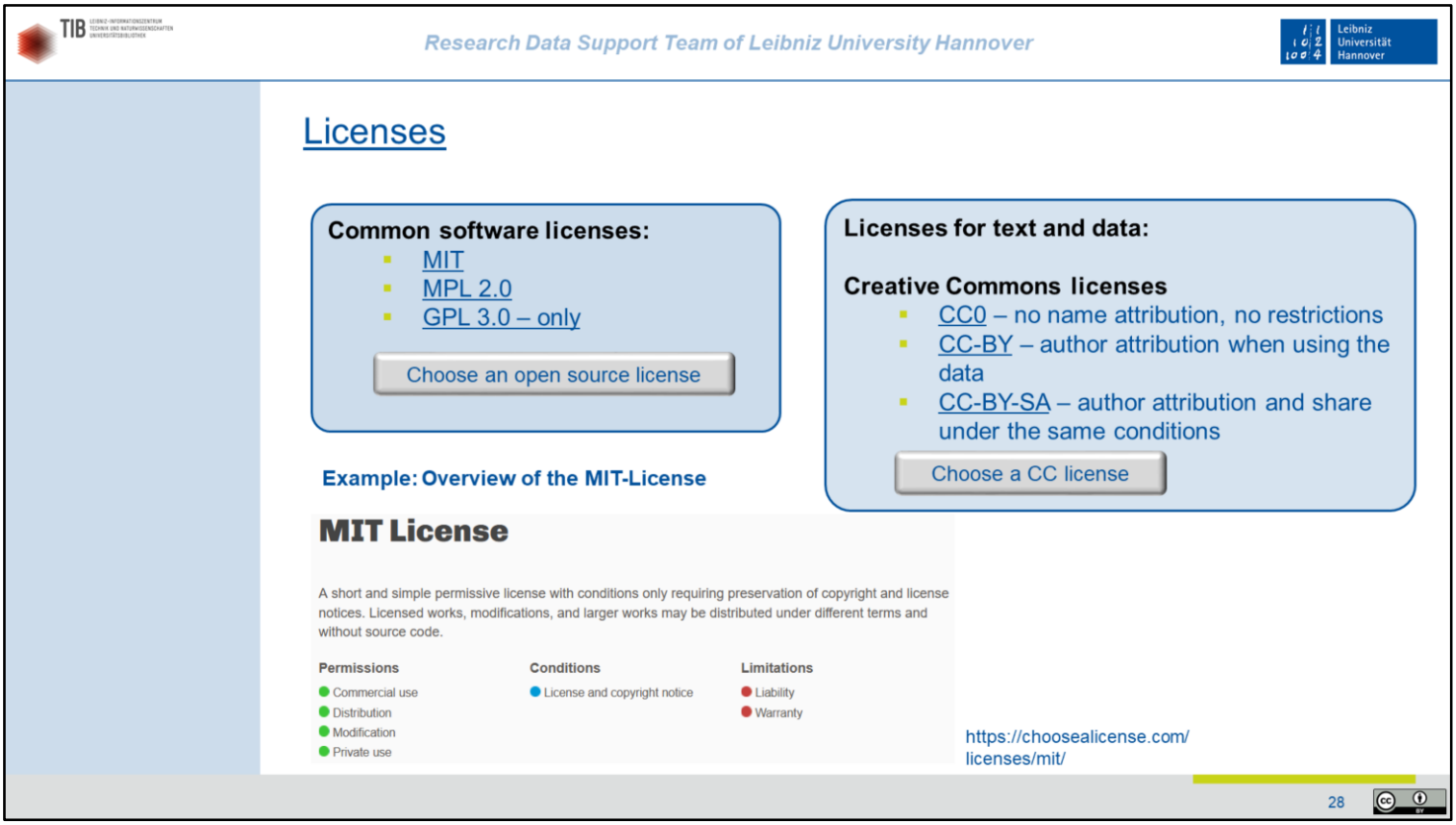
And secondly, you should provide citation metadata so that your software can be cited correctly. With a CFF file, which we already mentioned in one of the previous slides, this is quite easy to do.

Another important step: Choose a license under whose terms your software should be and draw attention to it, e.g. in a README file.

Source code repositories, such as GitHUb, are not suitable for publishing, by the way, because you won't get a DOI there, which is important if you want to cite and reference your data.

In addition to publishing in a research data repository, you can also publish your software in a peer-reviewed software journal to increase visibility.

You should always publish the software in a repository, then you can of course also publish it in a software journal or other paper.



As noted in the previous slide, when you publish your data, you usually need to license your dataset. Licenses clarify whether or not the published data can be reused and under what circumstances. It's important to know that very restricted licenses can hinder the reuse of your data by others. The most open licenses are CC0 and CC-BY. If you intend to combine and reuse different data sources, you must make sure that this is in accordance with the respective licenses of each data source.

Creative commons licenses: choose

When you make software code openly available, you need to provide a software license. In a similar way, the license provides the legal framework for the use, modification and sharing of the software and derived versions thereof. The open source initiative offers a catalogue of approved licenses which have undergone a review process. If you have no further experience, we recommend filtering for “Popular/strong community” licenses, since these are widely used and adapted.

Choose an open source software license

Summary

- Software management happens before you even start coding:
  - Clarify responsibilities
  - Agree on standards and methods and write them down (SMP)
  - Recognize that data and software management are work tasks that will consume some of your time
- Develop a culture that values data- and software management
  - Include the best practices in your everyday work
  - Think of your future self and anyone who might want to re-use your code
  - Engage in projects of others: Create and solve issues, reuse instead of setting up new

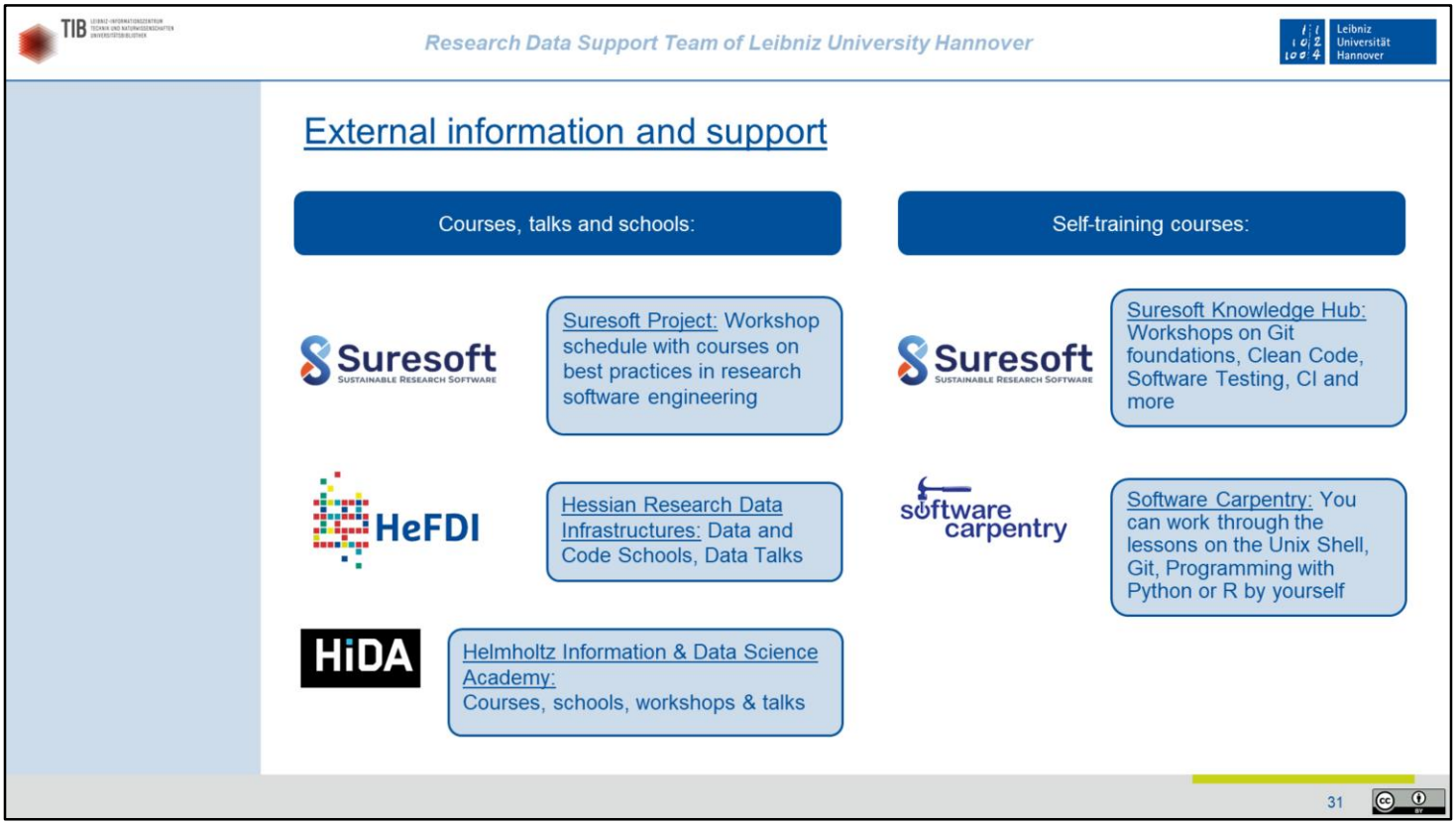
https://xkcd.com/1421/

We hope that you have found some inspiration how to make your software projects FAIR. There are many things you can do as an individual, but it is not only your responsibility to write and publish software in a clean and FAIR way. Group leaders need to emphasize these management practices to give guidance to the researchers. If you have the chance to foster awareness around the need for good software management practices, you can be a role model for others.

Supporting services and initiatives

- External information and support
- RSE working groups and initiatives

This course offers a overview over different aspects of software management. There are numerous sources of information and support that you can consult to go into more detail on some aspects that are interesting for you. In this final chapter, we would like to present some of them.



There is much more help and support beyond the university. Some organizations offer various courses and resources on research software management. Check the dates of the workshop schedule of the Suresoft Project, HeFDI and HiDA to take part in online workshops ore training schools.

Suresoft Workshop Schedule

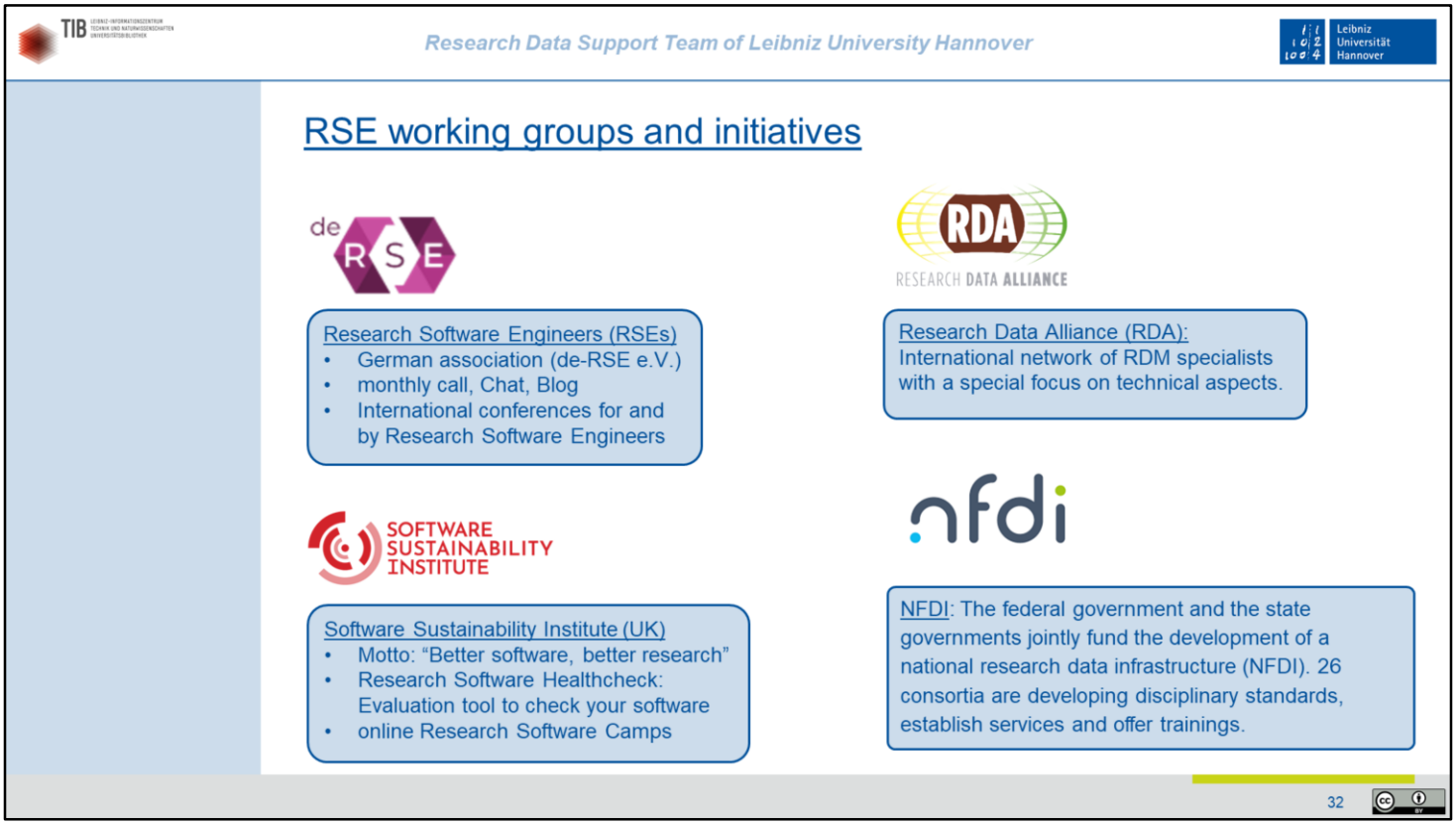
HeFDI Data Events

HiDA Course Catalog

You can also self-train with the workshops in the Suresoft Knowledge Hub or the lessons of the Software Carpentry.

Suresoft Knowledge Hub

Software Carpentry Lessons



There are cross-disciplinary initiatives and working groups which are working on new standards, workflows and trainings to help researchers to implement good research software development practices.

The Research Software Engineers (RSEs) are a community of researchers, scientists and others developing software in and for research within the German scientific landscape who want to make research software more sustainable and work towards changing the value RS has in the academic recognition and rewards system.

The Research Data Alliance focuses on infrastructure and data standards. The RDA is a global network with continental and national chapters and working groups for individual topics. The FAIR for Research Software (FAIR4RS) working group has published the FAIR Principles for Research Software (FAIR4RS Principles), which we have mentioned in this course.

The Software Sustainability Institute is based at the Universities of Edinburgh, Manchester, Oxford and Southampton, and experts with a breadth of experience in software development and training works towards the motto "Better software, better research".

Of particular importance to researchers are the disciplinary consortia that emerged in the scope of the national research data infrastructure, NFDI. These consortia receive long-term funding from the federal government and the states and are expected to establish disciplinary standards, to develop services and to organise the research community.

Thank you for taking this course!

by Auke Herrema

We wish you all the best for your future research data management!