Optical and Quantum Electronics (2021) 53:568
https://doi.org/10.1007/s11082-021-03192-4

Realization of best practices in software engineering and scientific writing through ready-to-use project skeletons

Michael Haider[1] · **Michael Riesch**[1] · **Christian Jirauschek**[1]

Received: 2 November 2020 / Accepted: 13 August 2021 / Published online: 4 September 2021

Abstract
Efforts in providing high-quality scientific software are hardly rewarded, as scientific output is typically measured in terms of publications in high ranking journals. As a result, scientific software is often developed without proper documentation and support of modern software design patterns. Ready-to-use project skeletons can be employed to accelerate the development process, while at the same time taking care of the implementation of best practices in software engineering. In this work, we revisit best practices in software engineering and review existing project skeletons. Special emphasis is given on the realization of best practices. Finally, we present a new project skeleton for scientific writing in LaTeX, which takes care of the attainment of best practices, adapted for being used in academic publications.

Keywords Software engineering · Scientific writing · Project skeleton · LaTeX

1 Introduction

Scientific software is frequently developed around well-established mathematical libraries that provide implementations of common algebraic and numerical methods. There are also more specialized packages that aim to simplify the numerical modeling and subsequent data analysis in scientific computing. Although these libraries and software packages are omnipresent in scientific software, acquiring funding for their continuous development and maintenance is notoriously difficult (Nowogrodzki 2019). Additionally, the time-consuming efforts put in the development of open-source toolboxes that form the basis of other scientific projects are hardly acknowledged.

The development of scientific software, may it be a general-purpose library to be used in other projects or a specific implementation to answer a certain scientific question,

This article is part of the Topical Collection on Numerical Simulation of Optoelectronic Devices.

Guest edited by Stefan Schulz, Silvano Donati, Karin Hinzer, Weida Hu, Slawek Sujecki, Alex Walker and Yuhrenn Wu.

✉ Michael Haider
michael.haider@tum.de

[1] Department of Electrical and Computer Engineering, Technical University of Munich, Arcisstr. 21, 80333 Munich, Germany

requires knowledge and formal training in both, the associated scientific discipline and software engineering in general (Nowogrodzki 2019; Wilson et al. 2014, 2017). Scientists from other fields than computer science occasionally lack the latter, which too often leads to poorly maintained software projects of at least questionable quality. However, there is plenty of literature available on how to produce good quality code, such as Hunt and Thomas (1999) on software engineering in general and Bangerth and Heister (2013), Prlić and Procter (2012), Wilson et al. (2014, 2017) in a more scientific computing oriented context. On top of that, the German Aerospace Center (DLR) (Schlauch et al. 2018) and the Netherlands eScience Center (Netherlands eScience Center 2019) provide a collection of development guidelines and best practices for the implementation of small to large-scale computational science projects. The former guideline distinguishes between four application classes, which reflect the scope and criticality of a planned software project, while the latter gives insights on best practices for different programming languages. However, those collections are exhaustive and intentionally kept general in order to apply to a wide range of software projects.

The final realization of best practices for a new software project comes along with tedious and time-consuming setup tasks. These tasks can be automatized in the context of ready-to-use project skeletons for a given programming language, where the skeleton takes care of the realization of most best practices, allowing scientists to focus more on the actual implementation. Of course, a project skeleton alone cannot implement all best practices as lined out in the comprehensive list by Schlauch et al. (2018). However, using predefined routines outlined by a project skeleton may reduce the level of required expertise in software engineering, while at the same time improving code-quality. As an example, we have already presented bertha (Riesch et al. 2020, 2020), an open-source project skeleton for C++ libraries with a Python interface.

The best practices from software engineering can also be applied to scientific writing in a slightly modified form. In many academic disciplines, scientific publications are typically typeset in LaTeX, which follows a coding cycle similar to writing software in a compiled programming language. Therefore, in terms of project management, coding style, independence of interests, and automation of repetitive tasks, scientific writing projects in LaTeX have similar characteristics to software engineering projects. Hence, one can compare the creation of a scientific LaTeX document, whether it is a large book or lecture project, or a small conferences abstract to a classical software design cycle. Both, maintaining a code base for scientific computing and the creation of articles, books, lecture notes, etc. in LaTeX start with a planning phase, in which the structure of the respective project is figured out. Afterward, the build system and compiler toolchains are set up, such that one arrives at a useable output, such as an executable in the case of software engineering or a printable document for scientific writing. Finally, there is the actual implementation phase, where the elements that have been planned are realized in terms of code, text, figures, snippets, etc. Arguably, the implementation phase in software engineering differs considerably from writing a scientific document, however, from a project management perspective, both can be treated equally as the stages where things get real. Additionally, depending on the size and scope of the respective project, there might be a long-term maintenance phase, which is also true for both, scientific writing and software engineering. Large and intermediate scientific writing projects are often authored by different people, possibly from different institutions. Hence, one requires some sort of project management, and one requires to agree on a common style in order to keep a document or a set of documents editable over an extended timeframe. When it comes to the development of new LaTeX packages, macros, classes and functions, which can indeed be considered to be a software engineering

project, also documentation, testing, and deployment become important. Hence, with the same reasoning, a ready-to-use project skeleton for scientific writing in LaTeX will be useful to facilitate the collaboration between researchers in the scope of scientific publications.

The present paper is organized as follows. After revisiting a non-exhaustive collection of best practices in software engineering in Sect. 2, we focus on the use of project skeletons like the aforementioned bertha project (Riesch et al. 2020) in scientific computing in Sect. 3. Finally, we introduce bertha-tex, a ready-to-use project skeleton for scientific writing in LaTeX, as suggested in (Riesch et al. 2020), in Sect. 4. In Sect. 5, the necessary steps for creating an instance of the bertha-tex project skeleton are outlined.

2 Best practices in software engineering

From the related literature (Bangerth and Heister 2013; Hunt and Thomas 1999; Netherlands eScience Center 2019; Nowogrodzki 2019; Prlić and Procter 2012; Wilson et al. 2014, 2017; Schlauch et al. 2018), we can summarize a list of 15 best practices in scientific software engineering, that can be grouped into seven categories (Riesch et al. 2020). The best practices and associated paradigms are language agnostic and can hence also be applied to software engineering as well as to scientific paper writing using LaTeX. This includes projects with the goal of creating a set of typesetting macros, i.e. LaTeX document classes and typesetting packages. In the following, we quickly go through the best practices that have been identified from the related literature. For a more exhaustive overview, see Riesch et al. (2020).

2.1 Project management

Project management is crucial, even for small software projects with only a single developer. Within the project management category, best practices include the use of a version control system, employing a collaboration platform, and agreeing on a specific workflow. A version control system (VCS) stores incremental changes to the source code of a project in a so-called project repository. The use of a version control system was independently recommended by all best practice guidelines that have been considered for the present paper, which highlights its importance. Furthermore, it is even recommended to use a VCS for small scripts that are intended for personal use only, regardless of the size and significance of the code. Among others, *Git* is a very prominent example of a VCS, which is regularly used and recommended by the authors of the present paper. We also use *Git* later on when introducing our project skeletons bertha and bertha-tex.

Along with a VCS, the use of a collaboration platform is recommended. Typically, web-based ticketing systems, also known as bug-tracking systems, are used to request code changes and report errors and mistakes in the project sources. The collaboration platform is the central project management tool, where the workload is distributed among the associated developers, while the archived change requests provide rudimentary documentation of discussions and design decisions that have been made in the course of the project. Collaboration platforms often come as all-in-one solutions with a possibility to host a version-controlled project repository with an associated issue tracking system and other related project management tools. We refer to collaboration platforms like *GitHub* or *GitLab* that provide a *git*-based VCS, allong with productivity tools for issue-tracking and documentation.

A collaboration platform together with a VCS, however, is just a set of tools gathered in the same place. To take full advantage of the tools and the VCS provided by the collaboration platform, it is essential to agree on a specific workflow, i.e. when and how a particular tool is used and to what extent. There are different paradigms and different recommendations for software engineering workflows that depend on the size and scope of the project. It is, however, beneficial to a project's success to agree on a respective workflow from the very beginning. The workflow, however, should be regularly reviewed and adapted to the current context if necessary. Often a scientific software or writing project starts in a small scope with only a limited number of contributors. If such a project scales later on, a more complex workflow can be chosen to fit the project's size and needs. For a specific collaboration platform, a workflow describes how issues are created and handled within the VCS in use. We suggest documenting the workflow in a specific *CONTRIBUTING.md* file in the root of your project's repository and would like to refer to established *Git*-workflows such as the GitHub flow or the GitLab flow (GitLab Inc. 2020).

2.2 Coding style

Depending on the specific programming language, the coding style, i.e. the formatting of the source code is most often irrelevant for the functionality of the built executables. However, following the paradigm "Write programs for people, not computers" (Wilson et al. 2014), it is the responsibility of the individual developer to produce easy-to-read and modular code. Especially but not exclusively for open-source projects, the source code needs to be seen as the developer's published work, similar to a scientific publication. As such, it should comply with the coding style of the whole project to form an easy-to-read and consistent source code, similar to a scientific paper that needs to be formatted according to a journal's style guide. The coding style typically includes two different things, the actual formatting of the source code, and language-specific styles and paradigms. The tedious task of maintaining a predefined coding style among several different source files, probably edited by a lot of different developers, can be automatized by means of so-called code formatting tools. Such tools are available for many different programming languages and, as such, also for the LaTeX typesetting system. However, one still needs to take care of appropriate function and variable names. In addition to the mere formatting of the code, it is also recommended to perform static code analysis, in order to avoid errors and bugs already while writing the source code. Tools that perform static code analysis are called linters. Linters find and highlight programming errors and bugs in the source code before the project is compiled into an executable. Most modern integrated development environments provide support for static code analysis for a variety of different programming languages.

2.3 Independence

Ideally, a project should be independent of any other interests. Thus, it is highly recommended to use open file formats and open-source libraries, unless there is a good reason not to. This not only concerns formats and libraries but also interpreters, compilers, and operating system support. The general recommendation is to support the most common operating systems and compiler toolchains in the respective domain.

Scientific software typically produces some numerical output data that needs to be stored for further post-processing. In the post-processing step, problem-specific scripts are

applied to the raw binary data to create e.g. visual representations of simulation results for further interpretation. In between the simulation and post-processing steps, which are to some degree independent from each other, the data is stored using a suitable file format. Some guidelines (Netherlands eScience Center 2019) recommend using open file formats, such as CSV or HDF5 for large data sets. This way, one can ensure that the results can be accessed independently of licensing and legal interests. For a lot of problems encountered during the development process, there are already well-established solutions in form of libraries and toolsets. However, some of these libraries are only distributed in binary form and often depend on restrictive licensing agreements. To ensure the operability of the code for an extended period of time, and to prevent vendor lock-in situations, it is highly recommended to rely on open-source libraries, as long as they provide a viable alternative to closed-source and commercial libraries and tools. This practice agrees well with the interoperability and reusability part of the FAIR principle (Lamprecht et al. 2020; Wilkinson et al. 2016) for scientific research software.

2.4 Automation

Repetitive tasks, such as building, testing, and deploying the software should be automatized as far as possible. Dependencies should be detected and dynamically linked in a platform-independent way. The platform-independent handling of dependencies can be accomplished using build automation tools, that provide the necessary build information for the respective platform. The software should be built and tested after each meaningful incremental change to the code, which can be performed automatically by means of continuous integration pipelines. Different platforms store common software libraries in different locations. It is recommended to use a build automation tool that detects whether a certain dependency is installed on the target platform, and if so, where the library is located on the system for the dynamic linking process. This way, one can maintain platform-independence through an additional build automation step. Here, each target platform is provided with the necessary build information in order to compile and link the code there. Each meaningful incremental code change should have an associated entry in the VCS, i.e. an associated *git commit*. Hence, build and test tasks can be triggered, as changes are committed to the project repository. This process is referred to as continuous integration. The description of the individual tasks that can be grouped into pipelines requires additional configuration, which depends on the continuous integration system in use. Ideally, the code is built and tested for all target platforms, and instant feedback is provided accordingly. It is recommended to set up continuous integration pipelines early in the development process, thereby detecting bugs and regressions effectively. Also, the deployment of the software can be automatized by means of continuous deployment. Here, specially marked versions of the software in the VCS are packaged and sent to a package repository that is either internal or publically available.

2.5 Documentation

The importance of documentation in scientific software projects cannot be highlighted enough (Bangerth and Heister 2013; Hunt and Thomas 1999; Netherlands eScience Center 2019; Nowogrodzki 2019; Prlić and Procter 2012; Wilson et al. 2014, 2017; Schlauch et al. 2018), especially if the scope of the project involves a broader audience. That said, good documentation involves descriptions for users and developers alike. The documentation

should be decomposed into different levels of abstraction, i.e. ”big-picture” documentation, presenting an overview of the project, and a detailed function reference that gives insights on how to use specific parts of the software.

A function reference is typically generated automatically from code comments using a specific annotation. The function reference should provide abstract documentation of classes and functions, where individual methods are seen as black boxes with respective inputs and outputs, regardless of the actual implementation. Regarding the actual implementation, simple code comments should be used to document the design and purpose of individual code snippets. One should refrain from commenting on simple mechanics and specific language constructs.

Within the ”big picture” documentation, an overview of the individual modules in the code should be given, that describes the larger scope of the software, including the aim of the project, installation notes, and dependencies. Additionally, for providing a clear history of changes, it is recommended to include a changelog that documents the features added to certain versions (Wilson et al. 2017).

2.6 Tests

Creating software is prone to errors and bugs. Thus, regular tests of individual modules of the source code help to improve the overall code quality. This requires that the code is structured into individual independent units that interact with each other. The effectiveness of tests can be monitored using code coverage tools, that create reports on which parts of the code are (not) covered by the applied test procedures. Depending on the programming language, there are different frameworks available which facilitate the creation of test routines for given chunks of code, called modules or units. As already mentioned in Subsect. 2.4, the execution of the test routines can be triggered by respective continuous integration pipelines, which ensure that code that is committed to the project repository is tested accordingly. As the writing of test routines for a certain module remains to be a manual task, it cannot be guaranteed that all individual parts of the source code are effectively tested. Therefore, it is useful to get an overview of the effectiveness of the tests performed by means of a so-called code coverage report. These reports can be generated by special code coverage tools (Schlauch et al. 2018).

2.7 Deployment

Depending on the scope of the project, also scientific software is often intended to be used by a larger community. Therefore, a way to distribute software packages to the users is often necessary, which should ideally be an established package repository (Nowogrodzki 2019). The necessary steps to create a ready-to-use package out of the bare source code involve building and bundling individual components. This step is platform-dependent and should be carried out using continuous deployment, as previously mentioned in Subsect. 2.4. This step automatizes the package creation for different platforms and pushes the resulting packaged software to a respective distribution environment.

As discussed in Subsect. 2.5, documentation is an integral part of a software project in the scope of scientific computing. The aforementioned function reference can be automatically generated within a continuous integration pipeline. The documentation, however, also needs to be deployed such that the target audience can access the respective documents. This is ideally accomplished through a project-specific website that hosts the project’s

documentation. Some collaboration platforms provide the option to host a project-specific website within the project repository. Publishing the online documentation is then also embedded in a continuous deployment pipeline.

3 Project skeletons for scientific software

In this section, we review existing project skeletons for common programming languages in scientific computing. The skeletons are investigated concerning the best practices in software engineering compiled in Sect. 2. Similar to Riesch et al. (2020), we consider three different types of scientific software projects. Highly optimized and performant code for numerical simulations is typically written in a compiled language such as C++, which constitutes our first exemplary project. Data analysis and visualization tasks, on the other hand, are most often implemented using an interpreted programming language such as Python, equipped with the respective modules. Hence, a Python project as an example of an interpreted language constitutes our second exemplary project. Finally, we consider a scientific writing project in LaTeX as our third and last example. For all three types of projects, we would like to review existing project skeleton approaches with respect to their implementation of best practices. It shall be noted, however, that there also exist project skeletons for other programming languages, such as MATLAB, GNU R, Java, etc. just to mention a few (Carré 2012; White 2021; Poizat 2020).

While for C++ and Python projects, there are several project skeletons publically available, there is, to the authors' best knowledge, not a single implementation available for scientific writing projects in LaTeX, that takes into account the best practices from Sect. 2. Due to this fact, we will present a project skeleton for scientific writing in LaTeX in Sect. 4. Regarding C++ projects, we consider the work by Kracejic (2015) as the most complete solution with respect to the implementation of best practices. For Python, the approach in Ioannides (2018) is very helpful. Apart from that, we have recently demonstrated the most comprehensive project skeleton for C++ with Python bindings (Riesch and Jirauschek 2019), which implements all best practices mentioned in Sect. 2, while it is also capable of building and installing an associated Python interface module using SWIG.

3.1 CleanCppProject (Kracejic 2015)

The cleanCppProject skeleton by Kracejic implements almost all best practices, apart from generating a code coverage report. This skeleton provides a formidable starting point for general purpose C++ projects in the scientific context. In Table 1, the respective implementations of best practices are listed. Note that the use of open file formats and open-source libraries depends on the respective instance of the skeleton. Hence, we have given some common recommendations in the respective rows. The same holds for online documentation and the aforementioned code coverage report. Table 1 and the following tables 2 and 3 are structured in the following way. The leftmost column provides a list of best practices from Sect. 2. The following column describes, how these best practices are implemented by the respective project skeleton under consideration. The rightmost column provides insight on how a user project as an instance of a skeleton is supposed to implement the respective recommendations in the first column.

Table 1 Implementations of best practices within the cleanCppProject skeleton by Kracejic

Best practice	C++ (Kracejic 2015)	User project
Version control system	Git	
Collaboration platform	GitLab, GitHub	
Workflow	GitLab Flow, GitHub Flow	
Code formatting tool	Clang-format	
Static code analysis	Clang-tidy	
Open file formats	User responsibility	e.g., JSON, CSV, HDF5
Open-source libraries	User responsibility	e.g., FFTW, GNU Scientific Library
Build automation	CMake	
Continuous integration	GitLab-CI, Travis CI	
Function reference	Doxygen	
Documentation	Markdown	
Unit test framework	Catch2	
Code coverage report	Not implemented	e.g., gcov
Deployment	CPack	
Online documentation	Not implemented	e.g., GitLab Pages, GitHub Pages

Table 2 Implementations of best practices within the Python Package Template Project skeleton by Ioannides

Best practice	Python (Ioannides 2018)	User project
Version control system	Git	
Collaboration platform	GitLab, GitHub	
Workflow	GitLab Flow, GitHub Flow	
Code formatting tool	Not implemented	e.g., black
Static code analysis	Not implemented	e.g., pylint
Open file formats	User responsibility	e.g., JSON, CSV, HDF5
Open-source libraries	User responsibility	e.g., FFTW, GNU Scientific Library
Build automation	Not required	
Continuous integration	Travis CI	
Function reference	Sphinx	
Documentation	ReStructuredText	
Unit test framework	Pytest	
Code coverage report	Not implemented	e.g., pytest-cov
Deployment	PyPI	
Online documentation	Not implemented	e.g., GitLab Pages, GitHub Pages

3.2 Python package template project (Ioannides 2018)

Starting off with a new Python project can be drastically simplified using the Python Package Template Project skeleton by Ioannides. The skeleton is distributed via PyPI, the Python Package Index, and features project management, continuous integration, documentation, unit testing, and deployment support. Overall, the project template takes care

Table 3 Implementations of best practices within the bertha project skeleton (Riesch et al. 2020)

Best practice	Bertha	User project
Version control system	Git	
Collaboration platform	GitLab, GitHub	
Workflow	GitLab Flow, GitHub Flow	
Code formatting tool	Clang-format	
Static code analysis	Clang-tidy	
Open file formats	User responsibility	e.g., JSON, CSV, HDF5
Open-source libraries	User responsibility	e.g., FFTW, GNU Scientific Library
Build automation	CMake	
Continuous integration	GitLab-CI, Travis CI	
Function reference	Doxygen	
Documentation	Markdown	
Unit test framework	Catch2	
Code coverage report	Gcov	
Deployment	Conda	
Online documentation	GitLab Pages, GitHub Pages	

of most of the tedious setup steps, which facilitates the realization of best practices for new Python projects. Additionally, we suggest the use of black (Python Software Foundation 2020) as a code formatting tool, together with pylint (Python Code Quality Authority 2020) for static code analysis. For code coverage report generation, the pytest-cov package can be used. We have summarized the implementation of best practices in the Python Package Template Project by Ioannides (2018) in Table 2, along with our additional recommendations for user projects.

3.3 Bertha (Riesch and Jirauschek 2019)

Rather than considering Python and C++ separately, the bertha project skeleton (Riesch et al. 2020) provides best practice implementations for a C++ library with Python bindings. Such a combination is quite common in scientific computing, as it combines the computational performance of C++ with the clarity and brevity of Python (Riesch et al. 2020). Within bertha, the focus lies on creating a highly performant library in C++ while the associated Python interface is generated automatically with the help of the SWIG (SWIG 2020) project. The bertha project skeleton implements all key elements from Sect. 2, which makes it the most comprehensive project skeleton for a standalone C++ project. Additionally, the quite intricate steps to build and install an associated Python interface module are implemented within CMake. Finally, the package is deployed via a conda feedstock (conda-forge 2019). The respective choices for the implementations of best practices in bertha are given in Table 3.

The bertha project skeleton has served as a template for creating an internal software project for the simulation of rapidly tunable Fourier domain mode-locked (FDML) fiber lasers (Jirauschek and Huber 2015, 2017). The permissive license of the skeleton also allows for internal projects that are not going to be published. Third-party packages, like scientific libraries or libraries for storing output data, can be conveniently installed using conda and are automatically detected by the CMake build system. The bertha template

originated from mbsolve Riesch et al. 2018; Riesch and Jirauschek 2017, 2021, an open-source solver for the Maxwell-Bloch equations (Jirauschek et al. 2019). Here, mbsolve serves as a reference implementation of best practices, that are handled by the bertha project skeleton.

4 Bertha-tex: project skeleton for scientific writing in LaTeX

To the authors' best knowledge, there is no extensive project skeleton available that handles and encourages the implementation of best practices in software engineering with respect to scientific writing in LaTeX. However, we believe that the availability of such a skeleton will facilitate the collaboration between researchers in the scope of scientific publications, especially for projects that involve researchers from different institutions. Therefore, we created bertha-tex, a project skeleton for scientific writing in LaTeX. An overview of the project skeleton is presented in Fig. 1. Similar to the skeletons that we have reviewed in Sect. 3, bertha-tex implements best practices in scientific software engineering that are adapted for being used in scientific writing. Similar to the previous section, where we have reviewed existing project skeletons in the scope of scientific

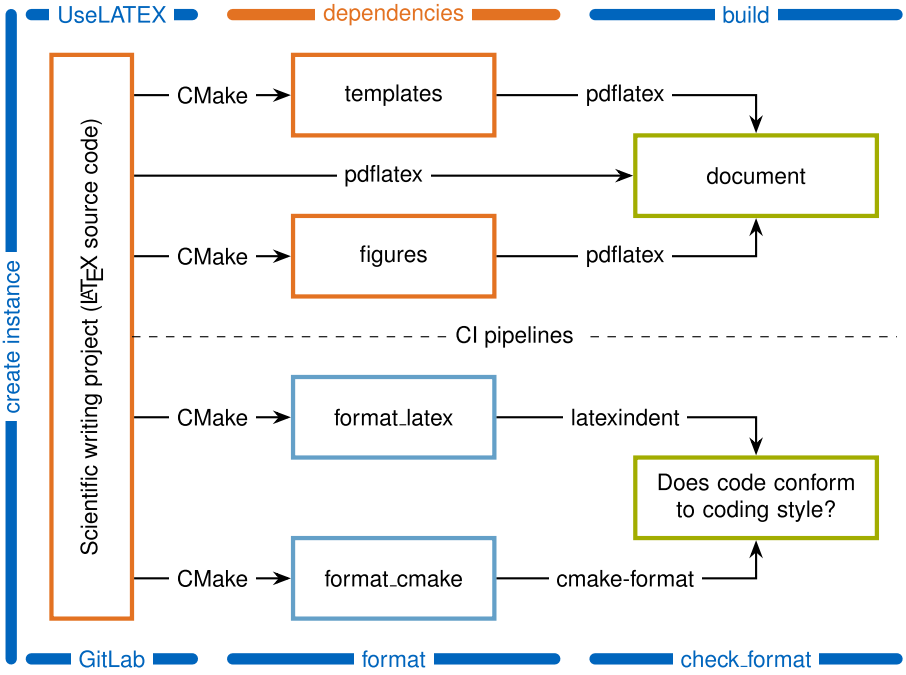

Fig. 1 Overview of the bertha-tex project skeleton. The source code and the respective dependencies are depicted using orange color. The dependencies are provided by the developers or authors, respectively. The project skeleton takes care of build and formatting steps. The CMake build system provides for different targets that either build or format the corresponding source code. The artifacts of the associated continuous integration pipelines (build, format) are marked using green color. The arrows connecting the individual blocks represent the individual tools that are being used

Table 4 Implementations of best practices within the bertha-tex project skeleton

Best practice	bertha-tex	User project
Version control system	Git	
Collaboration platform	GitLab, GitHub	
Workflow	GitLab Flow, GitHub Flow	
Code formatting tool	Latexindent	
Static code analysis	Lacheck	
Open file formats	User responsibility	e.g., JSON, CSV, HDF5
Open-source libraries	User responsibility	e.g., CTAN Packages, Templates
Build automation	CMake	
Continuous integration	GitLab-CI, Travis CI	
Function reference	Docstrip	
Documentation	LaTeX, Markdown	
Unit test framework	Not required	
Code coverage report	Not required	
Deployment	CTAN	
Online documentation	e.g., GitLab Pages, GitHub Pages	

software development, Table 4 presents an overview of the implementations of best practices and design choices for the bertha-tex skeleton.

Clearly, not all best practices described in Sect. 2 have a direct counterpart for all scientific writing processes, as we will explain in the following. While project management in terms of a version control system and a collaboration platform is certainly beneficial to any LaTeX codebase, the implementation of other best practices depends on the individual scope and reusability of the respective project. Scientific writing and general LaTeX projects can be categorized according to their lifetime and scope. We propose three different categories, i.e. short-lived, long-lived, and code-centered projects. Short-lived scientific writing projects include manuscripts and abstracts that are to be submitted to a journal or conference, as well as slides or posters for conference presentations. While there can be some sort of reusability, such writing projects typically end with the manuscript being submitted or revised/accepted, or the presentation being held. Note, however, that also short-lived publications might have a significant number of contributors, which necessitates efforts to enable collaborative writing. Long-lived scientific writing projects, on the other hand, distinguish themselves by a large degree of reusability. For example, a large book proposal, possibly involving different authors is an example for a long-lived project. Also lecture notes, presentations, exercise sheets, tutorial sheets, exams, etc., that are part of a course taught at a university belong to the second category, where one or several changing authors create content over timespans of multiple years. Finally, we have code-centered LaTeX projects, that can be more or less associated with standard software development. These include the development of new LaTeX classes, packages, and macros, with associated testing, documentation, and deployment. These categories, of course, have different scopes and needs for the implementation of best practices. With bertha-tex, however, we want to provide a common framework for all three categories, which means that we want to address as many best practices as possible. In the end, it is the user's choice to which extent the best practice implementations in the skeleton are finally made use of.

In recent years, web-based collaborative LaTeX editing and building tools, such as e.g. Overleaf (Writelatex Ltd. 2021) have become very popular. Those tools have a predefined build mechanism and integrate well with the *git* version control system. Hence, they naturally implement a lot of best practices for collaborative writing, especially for the first category of short-lived LaTeX projects. For larger-scale scientific writing projects that belong to the second category of long-lived projects, however, controlled build automation and increased performance through local builds become more important, which renders the use of web-based collaborative writing tools less suitable. Finally, for code-centered projects, such as the development of new LaTeX classes, online editors are completely inept. As soon as building multiple documents within a single project is desired, which might be the case for conference proceedings with an associated presentation, or when creating multiple documents like lecture notes, exercise sheets, etc., for a university course, the CMake build automation with continuous integration in bertha-tex is superior to other approaches.

The bertha-tex project skeleton for scientific writing in LaTeX is publically available (Haider et al. 2020) and can be used under the Apache 2.0 open-source license. In the following, we want to comment on the design choices that have led to the implementation of bertha-tex.

4.1 Project management

The bertha-tex project is hosted on the GitLab collaboration platform with a mirror repository on GitHub. Thus, we use the popular open-source git version control system together with the advanced project management tools of GitLab and GitHub, respectively. This is also encouraged for new scientific writing projects. Also if you do not intend to make the LaTeXmarkup code publically available, one can make use of private repositories within the respective platforms. As a workflow, we chose the GitLab Flow (GitLab Inc. 2020), which uses feature branches for the implementation of new features, where every non-trivial change starts with an issue in the associated issue tracking system.

4.2 Coding style

Within bertha-tex, we implemented code formatting as targets in CMake, which can handle formatting of LaTeX documents, packages, and classes through the open-source latexindent project (Hughes 2020) as well as formatting of the respective CMake files using cmake-format. The user can modify the predefined coding styles by editing the respective configuration file in the project source code. For static code analysis, we recommend lacheck.

4.3 Automation

Scientific writing projects supported by bertha-tex are built using the CMake build system, together with UseLATEX (Moreland 2020). A respective build pipeline has been implemented for being used with GitLab CI. Also, targets for code formatting of both CMake and LaTeX code have been made available within CMake. Subsequent continuous integration pipelines that check for compliance with the respective coding style are implemented as well.

4.4 Documentation

Currently, the bertha-tex project intends to provide a clean and solid basis for scientific paper writing in LaTeX. It enables the use of predefined templates, such that researchers can easily contribute to different scientific journals or conferences that typically require their own specific templates. Thus, there is currently no need for automated generation of documentation. However, bertha-tex can also be used for the development of new LaTeX packages, document classes, and macros, with only slight modifications in the CMake configuration. Therefore, automated documentation generation using docstrip shall be implemented in the near future.

4.5 Deployment

The bertha-tex project serves as a template for new scientific writing projects, and as such, it should be effectively deployed to the user. After the first stable version is released, distribution through the comprehensive TeX Archive Network is targetted.

5 Creating a skeleton instance

Starting a new paper is as simple as cloning bertha-tex using the mechanisms of GitLab or GitHub. Alternatively, the respective files can be copied manually into a new repository. The skeleton, however, might also be used as a starting point of more advanced academic writing projects, such as theses or books. It is also useful for scientific presentations, created with LaTeX beamer. Additionally, bertha-tex provides the required infrastructure for creating new packages, macros, and document classes in LaTeX. In the following, we restrict ourselves to the most common scenario of a user that wants to write a new academic paper. The necessary steps for this scenario can be decomposed into three stages that are briefly outlined hereafter.

5.1 Setup stage

After cloning or copying bertha-tex into a new and empty git repository that should host the LaTeX source code of the scientific article that is to be written, the project needs to be configured accordingly. Hence, it is useful to give the project a meaningful name, such as the abbreviation of the journal or conference the paper is to be submitted to, together with a keyword, describing the content of the article. The next step is to replace ”bertha-tex” with the given project name in all CMake configuration files. One might also want to rename the main document file ”bertha-tex.tex” accordingly. Then the LaTeX template, which is typically provided by the publisher, must be copied into the ”templates” directory, where it is automatically included by the CMake build system.

5.2 Writing stage

The actual document is created within the main document file. Includes and figures with external file references must be added to the respective CMake configuration file. The CMake build system then provides the necessary build information for the respective build environment. This is achieved by running CMake inside a special build directory, i.e.

```
mkdir -p build
cd build
cmake ..
```

One can then build the respective targets depending on the build-environment in use. There are specialized build targets called "format_latex" and "format_cmake", as well as a general target "format", which take care of the automated code formatting. Once the changes are pushed to the remote repository, continuous integration pipelines build the document and check the committed code for proper formatting.

5.3 Publication stage

As far as standalone documents are concerned, the deployment is typically handled by the publisher. The publication process is thus too diverse to create a continuous deployment pipeline that fits the needs of more than a single publisher. On the other hand, LaTeX packages, document classes, and macros are routinely distributed via the Comprehensive TeX Archive Network (CTAN). A continuous delivery pipeline for automated deployment via CTAN within bertha-tex shall be developed in the near future.

5.4 Examples and summary

Suppose now that we want to start writing a new article. We clone bertha-tex from (Haider et al. 2020) into a new blank *git* repository. Now we start by configuring the project name in the *README.md* and *CMakeLists.txt* files in the repository's root folder. In a next step, the template provided by the targetted journal is either installed within the LaTeX distribution or the respective LaTeX classes are copied to the templates directory. Finally, we rename the *bertha-tex.tex* in order to give our document a meaningful title and edit the respective *add_latex_document* entry in the *CMakeLists.txt* file. Now we are ready to start with writing the actual article. Building the article is done by creating a build folder and configuring the CMake project within it. Depending on the preferred CMake generator, the project targets can be built within this folder. It is useful to setup *lacheck* within a suitable editor for performing static code analysis. Code formatting is accomplished through running the `make format` command in the build directory. Figures and external references can be included and need to be referenced accordingly in the *CMakeLists.txt* file. In this way, dependencies are handled, i.e. as soon as an included file changes, the respective build job will be executed at the next build of the target.

If we now shift our focus away from creating a single article to maintaining a whole system of documents, as encountered e.g. when compiling documents for a university course,

we can finally make full use of the skeleton's features. In this case, each individual document has an associated build target in a *CMakeLists.txt* file, which can be distributed into several folders. In this way, figures and data visualizations can be reused among different documents, which are all built and maintained in a single code repository. For long term code maintenance, code formatting and checking pipelines provide for proper readability and reusability of code fragments. This means, that future changes that are incorporated into the project will be ensured to comply with the project's styleguide by the skeleton's predefined pipelines.

6 Conclusion

To this point, we have revisited a non-exhaustive list of best practices in software engineering in a language-agnostic form. The best practices then served as a benchmark for a review of existing project skeletons for both compiled and interpreted software. Within this review, we have presented bertha, our own project skeleton for C++ projects with Python bindings, where special emphasis was given to the fulfillment of all best practices that have been discussed in the beginning. As there are, however, various types of software projects without a corresponding skeleton, such as it was the case for a LaTeX project, we have introduced bertha-tex, a project skeleton for scientific writing. It should be noted, that the bertha-tex project skeleton is continuously developed. Thus, important features, like automated documentation generation and deployment to CTAN, will be implemented soon. Overall, the use of project skeletons facilitates the realization of best practices for software development as well as scientific writing. Project skeletons successfully help to overcome the barrier for implementing best practices by reducing the amount of knowledge needed and automating tedious setup steps for new projects.

Acknowledgements The authors would like to acknowledge the contributions of the LaTeX4EI project team to the bertha-tex project. A lot of the ideas and discussions from the TUM-Templates project influenced the creation of bertha-tex. Special thanks go to Michael Rinderle for stimulating discussions on the realization of the formatting pipeline.

Funding Open Access funding enabled and organized by Projekt DEAL.

Code availability The project skeletons *bertha* for C++ projects with Python bindings, as well as *bertha-tex* for scientific writing projects in LaTeX are publically available in the respective GitLab repositories: *bertha*: https://gitlab.com/cph-tum/bertha *bertha-tex*: https://gitlab.com/cph-tum/bertha-tex

Declarations

Conflicts of interest The authors declare that they have no conflict of interest.

References

Bangerth, W., Heister, T.: What makes computational open source software libraries successful? Comput. Sci. Disc. **6**, 015010 (2013). https://doi.org/10.1088/1749-4699/6/1/015010

Carré, J.B.: MathWorks MATLAB project template. https://github.com/speredenn/matlab-project-template (2012)

conda-forge: Conda feedstock for bertha. https://github.com/conda-forge/bertha-feedstock (2019)

GitLab Inc: Introduction to GitLab Flow. https://docs.gitlab.com/ee/topics/gitlab_flow.html (2020)

Haider, M., Riesch, M., Jirauschek, C.: bertha-tex: Project skeleton for scientific writing in LaTeX. https://gitlab.com/cph-tum/bertha-tex (2020)

Hughes, C.: latexindent.pl. https://github.com/cmhughes/latexindent.pl (2020)

Hunt, A., Thomas, D.: The Pragmatic Programmer: From Journeyman to Master, 1st edn. Addison-Wesley, Boston (1999)

Ioannides, A.: Python package template project for kick-starting new Python projects. https://github.com/AlexIoannides/py-package-template (2018)

Jirauschek, C., Huber, R.: Modeling and analysis of polarization effects in Fourier domain mode-locked lasers. Opt. Lett. **40**(10), 2385–2388 (2015). https://doi.org/10.1364/OL.40.002385

Jirauschek, C., Huber, R.: Efficient simulation of the swept-waveform polarization dynamics in fiber spools and Fourier domain mode-locked (FDML) lasers. J. Opt. Soc. Am. B **34**(6), 1135–1146 (2017). https://doi.org/10.1364/JOSAB.34.001135

Jirauschek, C., Riesch, M., Tzenov, P.: Optoelectronic device simulations based on macroscopic Maxwell-Bloch equations. Adv. Theor. Simul. **2**(8), 1900018 (2019). https://doi.org/10.1002/adts.201900018

Kracejic: Clean C++ project for you to use. https://github.com/kracejic/cleanCppProject (2015)

Lamprecht, A.L., Garcia, L., Kuzak, M., Martinez, C., Arcila, R., Martin Del Pico, E., Dominguez Del Angel, V., van de Sandt, S., Ison, J., Martinez, P.A., et al.: Towards FAIR principles for research software. Data Sci. **3**, 37–59 (2020). https://doi.org/10.3233/DS-190026

Moreland, K.: UseLATEX. https://gitlab.kitware.com/kmorel/UseLATEX (2020)

Netherlands eScience Center (2019) Software development guide. https://guide.esciencecenter.nl

Nowogrodzki, A.: How to support open-source software and stay sane. Nature **571**(7763), 133–134 (2019). https://doi.org/10.1038/d41586-019-02046-0

Poizat, P.: template-java-project. https://github.com/KentonWhite/ProjectTemplate (2020)

Prlić, A., Procter, J.B.: Ten simple rules for the open development of scientific software. PLoS Comput. Biol. **8**(12), e1002802 (2012). https://doi.org/10.1371/journal.pcbi.1002802

Python Code Quality Authority: pylint. https://github.com/PyCQA/pylint (2020)

Python Software Foundation: black. https://github.com/psf/black (2020)

Riesch, M., Jirauschek, C.: mbsolve: An open-source solver tool for the Maxwell-Bloch equations. https://github.com/mriesch-tum/mbsolve (2017)

Riesch, M., Jirauschek, C.: bertha: Project skeleton for scientific software (C++ with Python interface). https://gitlab.com/cph-tum/bertha (2019)

Riesch, M., Jirauschek, C.: mbsolve: An open-source solver tool for the Maxwell-Bloch equations. Comput. Phys. Commun. **4**, 108097 (2021).

Riesch, M., Tchipev, N., Senninger, S., Bungartz, H.J., Jirauschek, C.: Performance evaluation of numerical methods for the Maxwell-Liouville-von Neumann equations. Opt. Quant. Electron. **50**(2), 112 (2018). https://doi.org/10.1007/s11082-018-1377-4

Riesch, M., Haider, M., Jirauschek, C.: Project skeletons for scientific software. In: International Conference on Numerical Simulation of Optoelectronic Devices (NUSOD), pp 111–112 (2020). https://doi.org/10.1109/NUSOD49422.2020.9217756

Riesch, M., Nguyen, T.D., Jirauschek, C.: bertha: Project skeleton for scientific software. PLOS ONE **15**(3),e0230557 (2020). https://doi.org/10.1371/journal.pone.0230557

Schlauch, T., Meinel, M., Haupt, C.: DLR software engineering guidelines. https://doi.org/10.5281/zenodo.1344612 (2018)

SWIG: SWIG. https://github.com/swig/swig (2020)

White, K.: Projecttemplate. https://github.com/KentonWhite/ProjectTemplate (2021)

Wilkinson, M.D., Dumontier, M., Aalbersberg, I.J., Appleton, G., Axton, M., Baak, A., Blomberg, N., Boiten, J.W., da Silva Santos, L.B., Bourne, P.E., et al.: The FAIR guiding principles for scientific data management and stewardship. Sci. Data **3**, 160018 (2016). https://doi.org/10.1038/sdata.2016.18

Wilson, G., Aruliah, D.A., Brown, C.T., Chue Hong, N.P., Davis, M., Guy, R.T., Haddock, S.H.D., Huff, K.D., Mitchell, I.M., Plumbley, M.D., Waugh, B., White, E.P., Wilson, P.: Best practices for scientific computing. PLoS Biol. **12**(1), e1001745 (2014). https://doi.org/10.1371/journal.pbio.1001745

Wilson, G., Bryan, J., Cranston, K., Kitzes, J., Nederbragt, L., Teal, T.K.: Good enough practices in scientific computing. PLoS Comput. Biol. **13**(6), e1005510 (2017). https://doi.org/10.1371/journal.pcbi.1005510
Writelatex Ltd: Overleaf. https://github.com/overleaf/overleaf (2021)