Guidelines for Software Quality

CLARIAH Task 54.100

Maarten van Gompel
Centre for Language & Speech Technology
Radboud University, Nijmegen
Jauco Noordzij
Huygens ING
Reinier de Valk & Andrea Scharnhorst
Data Archiving and Networked Services (DANS)
Royal Netherlands Academy of Arts and Sciences

July 2, 2016

Revision 2

IMPORTANT NOTICE!

This is a **draft document**, everything in it is to be considered a **proposal only**! The guidelines have not been reviewed nor adopted yet!
Please see also the **Request for Comment** notice at the end of this document.

Contents

1 Motivation

CLARIAH aims to deliver a digital research infrastructure made explicitly accessible for researchers from the social sciences and humanities (SSH). This makes the development of advanced ICT tools a core activity within CLARIAH. To be able to assess the quality of the research infrastructure as a whole, we need to be able to assess the quality of its individual software parts, and their function inside of the research infrastructure including data components. If we can establish a common set of software guidelines, we may more readily identify weaker components of the software infrastructure and work on their improvement. We may also be able to better direct software production processes towards issues of interoperability and sustainability. A digital research infrastructure operates at the intersection of supporting ICT and research practices. For the SSH those are far from being homogeneous. The practices served by the infrastructure differ concerning the object of analysis, the processes of analysis, and the required usability level.

The need for increased attention to software quality and sustainability in an academic context is stressed by Doorn et al. [2016]. They recognize the fundamental role software plays in modern research and observe that good development practice is less followed in academia than in the commercial sector, which leads to problems in maintenance and adoption. They thus set out to promote good software development practice, good models of sustainability and dissemination of the best software across application areas/disciplines. We fully agree with these goals, and the guidelines presented here are intended to respond, on a practical level, to these with specific software quality criteria, explicitly encompassing sustainability as well.

1.1 Structure of the Document

After a discussion of the context of questions of software quality in social sciences and humanities research infrastructures, we start the main part of this document with the essentials: definitions and the state-of-art of discussions around software quality criteria. Following that, the guidelines—a number of actionable steps, intended to make software more usable and discoverable for users and other developers—are described. Two categories of guidelines are discerned:

- strict requirements
- additional guidelines

While the former are minimal requirements that software should adhere to, the latter constitute more generic guidelines that can help to achieve software of a higher quality. The requirements are specified in a form that leaves little room for discussion. They are but one way of meeting the more generic guidelines. It is encouraged to follow them as-is; doing so will make their implementation easier. It will also make consuming the results easier, as there will be a firm structure that can be relied on. The additional guidelines, on the other hand, are not simply boxes to tick. Instead, they offer ways to look at quality in software and tips to make software better and more useful. The guidelines are presented as open questions that can be answered in order to explain, for example, any trade-offs made while developing. It should be noted that throughout this document, both these terms are captured under the umbrella term *guidelines*. The aim of this document is, in an iterative process, to tailor the guidelines to the needs of the involved partners (ICT, research and archives).

2 Context

Software quality is an issue for software engineering. In the same way that experimental research depends on the quality of its instruments and the processes to use them properly, the quality of software influences the outcome of any computational processes: its accuracy (it should deliver what it was built for) as well as its efficiency (it should do this efficiently in time and with respect to use of resources). At a first glance, software quality concerns the quality of software code as a product (sometimes called its *functional quality*). However, software products do not exist in isolation, but they often form part of an architecture. So the quality of software concerns not only the level of source code, of a unit, but also its interplay with technology and systems (called *structural quality*). [1]

Software quality is a term that emerged within software engineering in the 1950's. The term *software sustainability*, however, is far less widespread. One could also say that with a more mature level of relying on software technologies, the aspect of their sustainability becomes more prevalent. The more software (and hardware) become seamless, invisible and move to the background, the more important becomes their quality also for use in the long term. The latter is one aspect of sustainability.

For infrastructures that aim to be reliable and supportive, the reliability and sustainability of elements they are built of is a must. For research infrastructures, an additional element that enhances the complexity emerges. Research practices by their very nature are changing in time. While we might rely on consolidated methods and theories to some extent—at least for the

[1] `https://en.wikipedia.org/wiki/Software_quality`

phases of normal science outside of the Kuhnian scientific revolutions—the research questions are supposed to change all the time, securely targeted towards the *unknown*. As a consequence, the application of methods and theories—their recombination (to use an evolutionary term)—is at the heart of research. For research based on the application of digital methods on digital material this implies a kind of constant revolution in software tools—those digital instruments. For science history that is actually not new. If we look into the history of instrumentation we will see outsourcing of instrument-making to industries and re-appropriating them for lab research and the tailor-made development of instruments both occurring at the same time. For an evaluation of software as part of a research infrastructure the differentiation between tools which are *explorative* and tools which are *stable and reproducible* is crucial. It makes sense to require quality assessment for both, but not all levels of the checklists presented in these guidelines might be of equal importance for all tools developed. To differentiate between them is one add-on task in developing these guidelines.

3 Introduction - glossary of terms

Assessing software (or data) quality is not a trivial matter.

Whenever we refer to *software*, we intend the term in a broad sense and encompassing all of the following aspects:

- source code
- binary executables
- user interfaces (including application programming interfaces (APIs) and web APIs)
- associated essential data
- documentation (including tutorials, screencasts)
- support infrastructure (version control, build systems, issue trackers)

The following guidelines for software quality consist of various assessment criteria, such as "Documentation" and "Installability". Each criterion in turn consists of assessment indicators formulated as a series of questions. This makes them directly applicable as an instrument for software quality assessment.

These guidelines target developers, managers and users of software. Developers will be more aware of the targets to meet, and able to identify and remedy weak areas. Managers and users will be able to assess whether software is of sufficient quality for their purposes. Although we intend to formulate the questions as plainly as possible, a certain degree of technical expertise is demanded of all of these in order to attain a successful assessment.

Each question, can be be answered with "yes", "no" or "not applicable" in case the guideline has no relevance for the software. The correct answer is always the affirmative. This will lead to measurable quality levels. Full compliance with the guidelines is achieved when all questions are answered affirmatively. In practice, however, this is not realistic to happen. Evaluators will need to determine for themselves what compliance level, per category, constitutes an acceptable passing threshold.

The criteria we yield are to a large extent adopted from the *Criteria-based Software Evaluation Guide* [Jackson et al., 2011] by the Software Sustainability Institute (SSI). Their work, in turn,

is modelled after *ISO/IEC 9126-1 Software Engineering - Product Quality* [ISO/IEC, 2001].

The Software Sustainability Institute[Crouch et al., 2013] is an academic institute explicitly geared towards researchers and software developers in science. The criteria they drafted are therefore deemed to form a very relevant basis for research projects such as CLARIAH. Software quality assessment will have a large generic component regardless of the context of the larger project. A large portion of these guidelines is therefore applicable in the broader context of software quality in academics, software quality for open source projects, and software quality in general.

The criteria used by the SSI are shown in Table 1.

Usability	
Understandability	Is the software easily understood?
Documentation	Comprehensive well-structured documentation?
Buildability	Straightforward to build from source on a supported system?
Installability	Straightforward to install and deploy on a supported system?
Learnability	Easy/intuitive to learn how to use its functions?
Sustainability & Manageability	
Identity	Project/software identity is clear and unique?
Copyright	Easy to see who owns the project/software?
Licencing	Adoption of appropriate license?
Governance	Easy to understand how the project is run and the development managed?
Community	Evidence of current/future community?
Accessibility	Evidence of good facilities to obtain versions of the software?
Testability	Easy to verify if the software functions correctly?
Portability	Usable on multiple platforms?
Supportability	Evidence of current/future developer support?
Analysability	Easy to understand at the source-code level?
Changeability	Easy to modify and contribute changes?
Evolvability	Evidence of current/future development?
Interoperability	Interoperable with other required/related software?

Table 1: Software quality criteria according to the SSI [Jackson et al., 2011] (with some minor paraphrasing).

We adopt this scheme with the following changes:

- Copyright & Licensing are merged.
- Evolvability was merged into Changeability.
- Security & Privacy criterion is added to Sustainability & Manageability
- Performance criterion was added to Usability.
- Reusability criterion was added to Sustainability & Manageability: To what extend is the software reusable?
- The Interoperability criterion is split into a generic section and a CLARIAH-specific section. The latter provides indicators to assess whether the software complies to requirements for integration into the CLARIAH infrastructure.

Sections 5 and 6 lay out all criteria with the individual assessment indicators. The individual indicators are inspired by the SSI guidelines but are not directly copied. Our indicators are

generally more condensed and may group issues that were expressed as multiple indicators in the SSI guide. Various indicators have also been added that did not exist in the SSI guide, and there were various indicators we found too specific to use.

Each indicator carries an ID code for easy reference. Cross-references will be made whenever there is a relationship between indicators.

Before the description of these more generic guidelines, a number of strict, minimal, requirements that software should adhere to (see Section 1.1), are described in the now following Section 4.

4 Minimal Requirements

This section describes a set of guidelines that are very straightforward to implement. They follow common practice where possible, and take a firm stand where interoperability benefits when a choice must be made. However, depending on the context, more (or less) may be expected from peers or collaborators.

It should be noted that there are two sets of opposing scenarios: the first concerns end user software versus experimental software, and the second software receiving active support versus unsupported software.

- Not all software is intended exclusively for **end users**: some software is **experimental**, and intended only for people who can modify the code. This makes such software less usable—but when the goal of a project is to verify if something is at all possible, this might be a valid trade-off.
- While everyone wants the software they use to receive **active support**, in practice this is not always feasible. It is acceptable to release **unsupported** software—provided that users are notified clearly and in advance.

Which of the following requirements apply depends on which configuration of these two sets of opposing scenarios is applicable. Below are written out all four configurations: in each case, non-applicable guidelines are grayed-out.

4.1 Configuration 1: Actively Supported End User Software

1. The software must be stored in a **version control system** (VCS). The source code must be published. To lower the barrier of entry the VCS must be either in a Git, Mercurial or SVN repository. As much as possible should be stored alongside the code in the VCS.
2. There must be **one command that installs all dependencies** on the supported operating systems (OSs).
3. The software must be **built with one command**. This command must not write to the file system outside of the source code directory and the temp directories (installation actions should be part of a separate script). Of course, the code will not be buildable on all OSs, and the command needs not to be platform independent—but there must be one OS on which it works without further configuration necessary. The command must call the multiple build steps if necessary. If there are tests, the command should run them as part of the build.

4. The software must either result in a directory with artefacts that can be moved to an arbitrary location on the system and still work, or it must contain **one command that will perform the installation**.
5. The software must be able to **run in the foreground** after installation.
6. All **runtime dependencies** must be available after installation.
7. If the software can be installed it must also have **one uninstall script**. This script must provide a switch to also remove config files and leave them alone by default.
8. The above requirements regarding installation can be omitted if the software can be run from the source directory and during running will not write outside of the source directory or temporary directories.
9. The root folder of the software project contains a file called **README** (fully capitilized), which must be a readable plain text file. It may link to images, but the reader must be able to follow along without them. A **readable plain text** file is a file that can be read by a human using both wordpad.exe and vi. Formats such as markdown and ReST are readable as plain text, but LaTeX and HTML are not. A UTF-8 encoding should be chosen; however, the use of ANSI, UTf-16 or UCS2 is also acceptable when forced by some technical constraint. (Though you must then mention that using the ascii subset at the start of the file itself.
    1. The readme must start by explicitly mentioning the experimental nature of the software.
    2. Accurate support contact information must be provided at the top of the readme, and an end date, until which this information can be assumed to be valid, must also be added here. A suitable end date is the current end time of the project. If no clear end date exists, it should be set no further than a year away, and it should be reset regularly.
    3. It must be mentioned explicitly that the software is unsupported
    4. The readme must contain an explanation of the intended audience of the software, where the different user groups should be clearly named and described.
    5. The readme must describe the problems that these users face, which resulted in the software.
    6. The readme must describe what the software does to help solving these problems.
    7. The readme must contain a list of similar or related software, and a description how the current software fits in.
    8. The readme must specify the commands used to build and install from point 2-6 above. Also, it should specifiy the OSes the commands will work on.
    9. The readme must contain either a quick start, explaining how the first task can be performed with the system, or, in the more extensive documentation, a link to the quick start.
    10. The readme must specify observed memory/file/processing usage while doing the happy path and a moderate load path.

11. The readme must contain a description of how the software scales (the load can be shared over multiple instances, explicitly mentioning if this is not possible).
12. The readme must contain a non-exhaustive list of places where the software is running.)
13. The readme must specify until when the project is currently funded, as well as a list of the funders.

10. Any **documentation must be stored alongside the code**. Long-form texts should live in the folder docs/ in the root of the project.
11. All **documentation must be also stored in an archivable format**. Such documentation can of course be generated from another source format, but the archivable version must also be stored in the VCS. An **archivable format** is a file format that will, with great probability, still be readable in ten years' time. Such formats are readable plain text (see above), HTML that does not depend on styles/js, and PDF/A. In the case of PDF/A, the source format must also be included—though it may be a non-archivable format such as a Word file, a LaTeX file, etc.
12. The project must have a **website**, containing, at the very least:
    (a) a typeset version of the readme
    (b) screenshots/screencasts of the application
    (c) a link to the source code
    (d) contact information, or a link to the bug tracker
    (e) a link to the documentation: either a direct link to the pdf or html, if available, or a browseable list of the text files
    (f) the website must be registered on Google, and a link must be provided to the project

    Note that the readme as typeset by GitHub fulfils almost all these criteria.
13. The **user interface of the application should point to the website**. If this user interface is a website, where possible, a link should be visible on every page. If the user interface is a command line app, a main page should be provided, or a text file that contains a link to the home page should be distributed with the binaries.
14. The application's **user interface should point to the contact information** or to a place to report bugs.
15. Almost every application provides an API. This **API must be discoverable**. For a Java project this means adding Javadoc where needed, and using descriptive names otherwise. For Javascript this means providing a document containing the API. What must be provided is a list of function names, their parameter names, and the constraints placed on their values. It would be helpful to know under what condition which function is expected to be called.
16. If the software contains a configuration file, then the root of the repository must contain a **configuration file containing all options** set to the sane defaults or commented out containing a description when you want to use them.

17. The software must be **distributed under an OSI approved licence** in a LICENSE file in root. Also copyright holder
18. The software must have a **CONTRIBUTING file**. (The template could be used—but it must be noted that that means one must be open to all PRs. Specifying some details in this file thus is very useful.
19. Data must be made available as **a log or rdf quads**.
20. The software must **not store usernames and passwords**; instead, it must use the instruments provided by the project.
21. Experimental software MUST NOT allow external users to store data (for this renders it production software).
22. This list of requirements must be **distributed alongside the code**.

4.2 Configuration 2: Unsupported End User Software

1. The software must be stored in a **version control system** (VCS). The source code must be published. To lower the barrier of entry the VCS must be either in a Git, Mercurial or SVN repository. As much as possible should be stored alongside the code in the VCS.
2. There must be **one command that installs all dependencies** on the supported operating systems (OSs).
3. The software must be **built with one command**. This command must not write to the file system outside of the source code directory and the temp directories (installation actions should be part of a separate script). Of course, the code will not be buildable on all OSs, and the command needs not to be platform independent—but there must be one OS on which it works without further configuration necessary. The command must call the multiple build steps if necessary. If there are tests, the command should run them as part of the build.
4. The software must either result in a directory with artefacts that can be moved to an arbitrary location on the system and still work, or it must contain **one command that will perform the installation**.
5. The software must be able to **run in the foreground** after installation.
6. All **runtime dependencies** must be available after installation.
7. If the software can be installed it must also have **one uninstall script**. This script must provide a switch to also remove config files and leave them alone by default.
8. The above requirements regarding installation can be omitted if the software can be run from the source directory and during running will not write outside of the source directory or temporary directories.
9. The root folder of the software project contains a file called **README** (fully capitilized), which must be a readable plain text file. It may link to images, but the reader must be able to follow along without them. A **readable plain text** file is a file that can be read by a human using both wordpad.exe and vi. Formats such as markdown and ReST are readable as plain text, but LaTeX and HTML are not. A UTF-8 encoding should be chosen; however, the use of ANSI, UTf-16 or UCS2 is also acceptable when forced by

some technical constraint. (Though you must then mention that using the ascii subset at the start of the file itself.

1. The readme must start by explicitly mentioning the experimental nature of the software.
2. Accurate support contact information must be provided at the top of the readme, and an end date, until which this information can be assumed to be valid, must also be added here. A suitable end date is the current end time of the project. If no clear end date exists, it should be set no further than a year away, and it should be reset regularly.
3. It must be mentioned explicitly that the software is unsupported
4. The readme must contain an explanation of the intended audience of the software, where the different user groups should be clearly named and described.
5. The readme must describe the problems that these users face, which resulted in the software.
6. The readme must describe what the software does to help solving these problems.
7. The readme must contain a list of similar or related software, and a description how the current software fits in.
8. The readme must specify the commands used to build and install from point 2-6 above. Also, it should specifiy the OSes the commands will work on.
9. The readme must contain either a quick start, explaining how the first task can be performed with the system, or, in the more extensive documentation, a link to the quick start.
10. The readme must specify observed memory/file/processing usage while doing the happy path and a moderate load path.
11. The readme must contain a description of how the software scales (the load can be shared over multiple instances, explicitly mentioning if this is not possible).
12. The readme must contain a non-exhaustive list of places where the software is running.)
13. The readme must specify until when the project is currently funded, as well as a list of the funders.

10. Any **documentation must be stored alongside the code**. Long-form texts should live in the folder docs/ in the root of the project.
11. All **documentation must be also stored in an archivable format**. Such documentation can of course be generated from another source format, but the archivable version must also be stored in the VCS. An **archivable format** is a file format that will, with great probability, still be readable in ten years' time. Such formats are readable plain text (see above), HTML that does not depend on styles/js, and PDF/A. In the case of PDF/A, the source format must also be included—though it may be a non-archivable format such as a Word file, a LaTeX file, etc.
12. The project must have a **website**, containing, at the very least:
    (a) a typeset version of the readme

(b) screenshots/screencasts of the application

(c) a link to the source code

(d) contact information, or a link to the bug tracker

(e) a link to the documentation: either a direct link to the pdf or html, if available, or a browseable list of the text files

(f) the website must be registered on Google, and a link must be provided to the project

Note that the readme as typeset by GitHub fulfils almost all these criteria.

13. The **user interface of the application should point to the website**. If this user interface is a website, where possible, a link should be visible on every page. If the user interface is a command line app, a main page should be provided, or a text file that contains a link to the home page should be distributed with the binaries.

14. The application's **user interface should point to the contact information** or to a place to report bugs.

15. Almost every application provides an API. This **API must be discoverable**. For a Java project this means adding Javadoc where needed, and using descriptive names otherwise. For Javascript this means providing a document containing the API. What must be provided is a list of function names, their parameter names, and the constraints placed on their values. It would be helpful to know under what condition which function is expected to be called.

16. If the software contains a configuration file, then the root of the repository must contain a **configuration file containing all options** set to the sane defaults or commented out containing a description when you want to use them.

17. The software must be **distributed under an OSI approved licence** in a LICENSE file in root. Also copyright holder

18. The software must have a **CONTRIBUTING file**. (The template could be used—but it must be noted that that means one must be open to all PRs. Specifying some details in this file thus is very useful.

19. Data must be made available as **a log or rdf quads**.

20. The software must **not store usernames and passwords**; instead, it must use the instruments provided by the project.

21. Experimental software MUST NOT allow external users to store data (for this renders it production software).

22. This list of requirements must be **distributed alongside the code**.

4.3 Configuration 3: Actively Supported Experimental Software

1. The software must be stored in a **version control system** (VCS). The source code must be published. To lower the barrier of entry the VCS must be either in a Git, Mercurial or SVN repository. As much as possible should be stored alongside the code in the VCS.

2. There must be **one command that installs all dependencies** on the supported operating systems (OSs).
3. The software must be **built with one command**. This command must not write to the file system outside of the source code directory and the temp directories (installation actions should be part of a separate script). Of course, the code will not be buildable on all OSs, and the command needs not to be platform independent—but there must be one OS on which it works without further configuration necessary. The command must call the multiple build steps if necessary. If there are tests, the command should run them as part of the build.
4. The software must either result in a directory with artefacts that can be moved to an arbitrary location on the system and still work, or it must contain **one command that will perform the installation**.
5. The software must be able to **run in the foreground** after installation.
6. All **runtime dependencies** must be available after installation.
7. If the software can be installed it must also have **one uninstall script**. This script must provide a switch to also remove config files and leave them alone by default.
8. The above requirements regarding installation can be omitted if the software can be run from the source directory and during running will not write outside of the source directory or temporary directories.
9. The root folder of the software project contains a file called **README** (fully capitilized), which must be a readable plain text file. It may link to images, but the reader must be able to follow along without them. A **readable plain text** file is a file that can be read by a human using both wordpad.exe and vi. Formats such as markdown and ReST are readable as plain text, but LaTeX and HTML are not. A UTF-8 encoding should be chosen; however, the use of ANSI, UTf-16 or UCS2 is also acceptable when forced by some technical constraint. (Though you must then mention that using the ascii subset at the start of the file itself.
    1. The readme must start by explicitly mentioning the experimental nature of the software.
    2. Accurate support contact information must be provided at the top of the readme, and an end date, until which this information can be assumed to be valid, must also be added here. A suitable end date is the current end time of the project. If no clear end date exists, it should be set no further than a year away, and it should be reset regularly.
    3. It must be mentioned explicitly that the software is unsupported
    4. The readme must contain an explanation of the intended audience of the software, where the different user groups should be clearly named and described.
    5. The readme must describe the problems that these users face, which resulted in the software.
    6. The readme must describe what the software does to help solving these problems.
    7. The readme must contain a list of similar or related software, and a description how the current software fits in.

8. The readme must specify the commands used to build and install from point 2-6 above. Also, it should specifiy the OSes the commands will work on.
9. The readme must contain either a quick start, explaining how the first task can be performed with the system, or, in the more extensive documentation, a link to the quick start.
10. The readme must specify observed memory/file/processing usage while doing the happy path and a moderate load path.
11. The readme must contain a description of how the software scales (the load can be shared over multiple instances, explicitly mentioning if this is not possible).
12. The readme must contain a non-exhaustive list of places where the software is running.)
13. The readme must specify until when the project is currently funded, as well as a list of the funders.

10. Any **documentation must be stored alongside the code**. Long-form texts should live in the folder docs/ in the root of the project.
11. All **documentation must be also stored in an archivable format**. Such documentation can of course be generated from another source format, but the archivable version must also be stored in the VCS. An **archivable format** is a file format that will, with great probability, still be readable in ten years' time. Such formats are readable plain text (see above), HTML that does not depend on styles/js, and PDF/A. In the case of PDF/A, the source format must also be included—though it may be a non-archivable format such as a Word file, a LaTeX file, etc.
12. The project must have a **website**, containing, at the very least:
    (a) a typeset version of the readme
    (b) screenshots/screencasts of the application
    (c) a link to the source code
    (d) contact information, or a link to the bug tracker
    (e) a link to the documentation: either a direct link to the pdf or html, if available, or a browseable list of the text files
    (f) the website must be registered on Google, and a link must be provided to the project

    Note that the readme as typeset by GitHub fulfils almost all these criteria.
13. The **user interface of the application should point to the website**. If this user interface is a website, where possible, a link should be visible on every page. If the user interface is a command line app, a main page should be provided, or a text file that contains a link to the home page should be distributed with the binaries.
14. The application's **user interface should point to the contact information** or to a place to report bugs.
15. Almost every application provides an API. This **API must be discoverable**. For a Java project this means adding Javadoc where needed, and using descriptive names

otherwise. For Javascript this means providing a document containing the API. What must be provided is a list of function names, their parameter names, and the constraints placed on their values. It would be helpful to know under what condition which function is expected to be called.

16. If the software contains a configuration file, then the root of the repository must contain a **configuration file containing all options** set to the sane defaults or commented out containing a description when you want to use them.
17. The software must be **distributed under an OSI approved licence** in a LICENSE file in root. Also copyright holder
18. The software must have a **CONTRIBUTING file**. (The template could be used—but it must be noted that that means one must be open to all PRs. Specifying some details in this file thus is very useful.
19. Data must be made available as **a log or rdf quads**.
20. The software must **not store usernames and passwords**; instead, it must use the instruments provided by the project.
21. Experimental software MUST NOT allow external users to store data (for this renders it production software).
22. This list of requirements must be **distributed alongside the code**.

4.4 Configuration 4: Unsupported Experimental Software

1. The software must be stored in a **version control system** (VCS). The source code must be published. To lower the barrier of entry the VCS must be either in a Git, Mercurial or SVN repository. As much as possible should be stored alongside the code in the VCS.
2. There must be **one command that installs all dependencies** on the supported operating systems (OSs).
3. The software must be **built with one command**. This command must not write to the file system outside of the source code directory and the temp directories (installation actions should be part of a separate script). Of course, the code will not be buildable on all OSs, and the command needs not to be platform independent—but there must be one OS on which it works without further configuration necessary. The command must call the multiple build steps if necessary. If there are tests, the command should run them as part of the build.
4. The software must either result in a directory with artefacts that can be moved to an arbitrary location on the system and still work, or it must contain **one command that will perform the installation**.
5. The software must be able to **run in the foreground** after installation.
6. All **runtime dependencies** must be available after installation.
7. If the software can be installed it must also have **one uninstall script**. This script must provide a switch to also remove config files and leave them alone by default.

8. The above requirements regarding installation can be omitted if the software can be run from the source directory and during running will not write outside of the source directory or temporary directories.

9. The root folder of the software project contains a file called **README** (fully capitilized), which must be a readable plain text file. It may link to images, but the reader must be able to follow along without them. A **readable plain text** file is a file that can be read by a human using both wordpad.exe and vi. Formats such as markdown and ReST are readable as plain text, but LaTeX and HTML are not. A UTF-8 encoding should be chosen; however, the use of ANSI, UTf-16 or UCS2 is also acceptable when forced by some technical constraint. (Though you must then mention that using the ascii subset at the start of the file itself.

    1. The readme must start by explicitly mentioning the experimental nature of the software.
    2. Accurate support contact information must be provided at the top of the readme, and an end date, until which this information can be assumed to be valid, must also be added here. A suitable end date is the current end time of the project. If no clear end date exists, it should be set no further than a year away, and it should be reset regularly.
    3. It must be mentioned explicitly that the software is unsupported
    4. The readme must contain an explanation of the intended audience of the software, where the different user groups should be clearly named and described.
    5. The readme must describe the problems that these users face, which resulted in the software.
    6. The readme must describe what the software does to help solving these problems.
    7. The readme must contain a list of similar or related software, and a description how the current software fits in.
    8. The readme must specify the commands used to build and install from point 2-6 above. Also, it should specifiy the OSes the commands will work on.
    9. The readme must contain either a quick start, explaining how the first task can be performed with the system, or, in the more extensive documentation, a link to the quick start.
    10. The readme must specify observed memory/file/processing usage while doing the happy path and a moderate load path.
    11. The readme must contain a description of how the software scales (the load can be shared over multiple instances, explicitly mentioning if this is not possible).
    12. The readme must contain a non-exhaustive list of places where the software is running.)
    13. The readme must specify until when the project is currently funded, as well as a list of the funders.

10. Any **documentation must be stored alongside the code**. Long-form texts should live in the folder docs/ in the root of the project.

11. All **documentation must be also stored in an archivable format**. Such documentation can of course be generated from another source format, but the archivable version must also be stored in the VCS. An **archivable format** is a file format that will, with great probability, still be readable in ten years' time. Such formats are readable plain text (see above), HTML that does not depend on styles/js, and PDF/A. In the case of PDF/A, the source format must also be included—though it may be a non-archivable format such as a Word file, a LaTeX file, etc.
12. The project must have a **website**, containing, at the very least:
    (a) a typeset version of the readme
    (b) screenshots/screencasts of the application
    (c) a link to the source code
    (d) contact information, or a link to the bug tracker
    (e) a link to the documentation: either a direct link to the pdf or html, if available, or a browseable list of the text files
    (f) the website must be registered on Google, and a link must be provided to the project

    Note that the readme as typeset by GitHub fulfils almost all these criteria.
13. The **user interface of the application should point to the website**. If this user interface is a website, where possible, a link should be visible on every page. If the user interface is a command line app, a main page should be provided, or a text file that contains a link to the home page should be distributed with the binaries.
14. The application's **user interface should point to the contact information** or to a place to report bugs.
15. Almost every application provides an API. This **API must be discoverable**. For a Java project this means adding Javadoc where needed, and using descriptive names otherwise. For Javascript this means providing a document containing the API. What must be provided is a list of function names, their parameter names, and the constraints placed on their values. It would be helpful to know under what condition which function is expected to be called.
16. If the software contains a configuration file, then the root of the repository must contain a **configuration file containing all options** set to the sane defaults or commented out containing a description when you want to use them.
17. The software must be **distributed under an OSI approved licence** in a LICENSE file in root. Also copyright holder
18. The software must have a **CONTRIBUTING file**. (The template could be used—but it must be noted that that means one must be open to all PRs. Specifying some details in this file thus is very useful.
19. Data must be made available as **a log or rdf quads**.
20. The software must **not store usernames and passwords**; instead, it must use the instruments provided by the project.

21. Experimental software MUST NOT allow external users to store data (for this renders it production software).
22. This list of requirements must be **distributed alongside the code**.

5 Quality Assessment Criteria - Usability

5.1 Understandability

U1 – Is it clear what the software does?

Software must be accompanied be a clear and concise high-level description, describing what exactly it does. Both the `README` file that ships with the software (see IS7) as well as the project website (see ID2) should contain this information.

U2 – Is it clear for whom the software is intended?

It should be clear who are the intended users for the software. Software is usually not appropriate for all audiences. Gearing software at multiple audiences however, through for instance offering multiple interfaces (command line interface (CLI) , graphical user interface (GUI), web-user interface (WUI), web service) is good practice. References to projects already using the software are recommended (see also CM1).

U3 – Is it clear how the software works?

There should be a high-level description explaining how the software accomplishes its task. Links to publications are recommended. Also, a schema offering an architectural overview is suggested where appropriate.

U4 – Is the software motivated?

There should be a written motivation for why the software does things the way it does and why it was designed in the first place. It should be clear what problems are solved by it. Links to publications and comparisons to similar software are strongly recommended.

5.2 Documentation

When we refer to documentation, we refer to the set of all documentation available for the software. This may consists of different types of documentation for different audiences, and may include published papers.

D1 – Is there documentation?

All software should be properly documented. Software without any documentation is as good as useless. At the very least, there should be some documentation at a minimum level, targeted

at the intended audience. A README (see IS7) that attempts to meet some of guidelines in this section (such as D6,D8,D9) can be considered a bare minimum level of documentation.

D2 – Is the documentation accessible?

Documentation must be publicly accessible and in an acceptable standard format such as HTML or PDF.

D3 – Is the documentation clear?

Documentation should be written in clear language, use proper spelling, and clearly describe the software. Step-by-step and task-oriented instructions are recommended.

See also D7.

D4 – Is the documentation complete?

Documentation should cover the entire software, including advanced features. Features that are not documented in any way are generally useless. If the software consists of multiple tools, are all documented? Are there no tasks the software can perform that are not properly explained?

D5 – Is the documentation accurate?

Documentation should describe the advertised version and not be out-of-date with the latest release. Examples should be in line with how the tool looks and behaves.

D6 – Does the documentation provide a high-level overview of the software?

A high-level overview of the software should be an integral part of the documentation, alongside more detailed instruction where appropriate. Documentation should not immediately dive into the details. It is important to first provide users with the necessary high-level insights so they can understand how these details form a part of the larger picture.

D7 – Are all the necessary audiences addressed, at their appropriate levels?

Different groups of users require different documentation. Developers require APIs if the software is a library (see L4), end-users require a walkthrough of the GUI if the software has one. A different level of expertise may be expected of different user groups, the documentation should assume the appropriate level.

D8 – Does the documentation make use of adequate examples?

Documentation should contain examples appropriate for the interface that is described. Command-line interfaces should see examples of invocation and input and output. Graphical user interfaces should be illustrated through screenshots or screencasts. API references should contain source code examples of usage.

D9 – Is there adequate troubleshooting information?

Documentation should include information on troubleshooting, i.e. a specification of possible error messages and explanation for resolution. A frequently asked questions (FAQ) section is appropriate to cover questions that are repeatedly asked by the user base.

D10 – Is the documentation available from the project website?

Documentation must be clearly linked from the project website.

D11 – Is the documentation under version control?

The sources for documentation must be under version control like the source code, preferably alongside the code itself.

5.3 Learnability

This category partially overlaps with the documentation criterion, but explicitly focusses on how straightforward it is to learn to use the software.

L1 – Is there a Getting started guide?

A *Getting started* guide outlines how to quickly get started with the software using a basic example.

L2 – Are there instructions for basic use cases?

Instructions should be provided for at least basic use cases, and ideally for all possible use cases if this is feasible.

L3 – Does the interface provide a help reference?

Help options should be provided by the interface. Command line interfaces must have a -h/–help pair describing usage and all options. GUIs should use tooltips/hints to clarify their widgets (or through whatever convention is customary for the platform/ecosystem). Alternatively, they can provide a help option referring to the documentation.

L4 – Is there API documentation for developers?

If the software is a programming library, API documentation must be provided. If the software is a web service, a specification of the web API must be provided.

L5 – If the software is configurable, are the configuration options clearly explained?

If the software is configurable, through for instance an external configuration files, a preferences window in a GUI, or by any other means, then the configuration options and their effect should be clearly documented.

Moreover, default values for configuration parameters, and their effect, should be clearly explained.

5.4 Buildability

Buildability applies to all software written in languages that compile to either native machine code or any intermediate byte code to be interpreted by a VM. This is contrasted to software that is interpreted at run-time from source code. This section is therefore applicable only to languages such as C, C++, Java, Pascal, Haskell, Scala, Rust, Cython but not to scripting language such as Python, Ruby, Perl, Go. Note that buildability does not include installation, packaging, or deployment.

B1 – Are there good instructions for building/compiling the software?

Build/compilation instruction should be available and clear enough. They should be distributed alongside the software's source distribution (as part of an `INSTALL` file or `README`), and/or be addressed in the documentation. If the source distribution is the primary means of distribution, then build instructions should be prominently displayed on the project website as well.

B2 – Is an established automated build system used?

Established build systems should be used. For example the GNU Build System[2] or CMake for C/C++ (this is preferred over a static Makefile); or Ant or Maven for Java.

Solutions that are not tied to a single IDE are always preferred.

B3 – Are all dependencies listed and available?

(related to IS2)

All required or optional dependencies should be listed, including those by third parties (with references to their websites). If the build system supports automatically obtaining dependencies, then this is a preferred solution. If the platform has a package manager that can install these dependencies, then add instructions (i.e. package names) to accomplish this.

[2] Also known as the autotools.

Moreover, all listed dependencies should be available, unobtainable software can not be used as a dependency. Higher-order dependencies need not be listed.

B4 – Are there tests to verify the build has succeeded?

A build process should terminate with a testing stage that verifies its success.

5.5 Installability

Installability concerns the deployment of software on the target platform, also including configuration of the software to the user's needs.

IS1 – Are there easily accessible installation instructions?

All software must come with installation instructions. Those instructions should be easily accessible and presented on the project website as well as shipped with the software in the README or INSTALL file (see also IS7). Build and installation instructions can be combined if the software is published as a source distribution.

IS2 – Are all dependencies listed and available?

(related to B3)

All required or optional dependencies should be listed, including those by third parties (with references to their websites). If the installation procedure supports automatically obtaining dependencies, then this is a preferred solution (see also IS3). If the platform has a package manager that can install these dependencies, then add instructions (i.e. package names) to accomplish this.

Moreover, all listed dependencies should be available, unobtainable software can not be used as a dependency. Higher-order dependencies need not be listed.

IS3 – Are programming language's best installation practices followed?

The ecosystem surrounding programming languages may come with standard procedures for installation. Software should comply to these rather than use ad-hoc mechanisms. If the ecosystem has facilities for dependency management (automatic download of dependencies), these should be used.

- Python software should have a `setup.py` based on Distutils, Setuptools, or their relatives.
- Installation of C/C++ software is generally an extension of the build process, often culminating in a `make install`.
- Java software is delivered as a `jar` file.
- Perl software should use systems such as `Module::Build`, `ExtUtils::MakeMaker`, or `Module::Install`. Resulting in a `Build.PL` or `Makefile.PL`.

Complying to these installation practices often means your installable package is fit for inclusion into the programming language's public package repository (see AC5).

IS4 – Is the software packaged according to standards for the target platform?

If the software has a well-defined target platform and is intended for adoption by a wider audience, it is recommended to package it in the appropriate form. This may however conflict with IS3, be rendered obsolete by IS3, or render IS3 obsolete. The decision whether to use packaging methods for the target platform and/or packaging methods for the ecosystem surrunding the programming language (IS3) should be made on an individual basis, considering the nature and audience of the software.

- Linux/BSD/Unix – Distributions generally have their own package manager
  - Arch Linux: Use the Arch User Repository (AUR)
  - Debian/Ubuntu (and other derivates): Use Debian Packages (deb)
  - RedHat/CentOS/Fedora (and other derivates): Use Red Hat Packages (rpm)
- Mac OS X: Use PKG or DMG for traditional Mac applications intended for end-users; for the more Unix-style software, use a system such as Homebrew or MacPorts
- Android: Use Android Application Packages (APK)
- iOS: (TODO)
- Windows: (TODO)

It is not always time- and cost-effective to package the software, especially not when there are multiple target platforms.

IS5 – Is the software package properly structured?

The contents of the package should be properly organised in sub-directories (for documentation, headers, source, etc..). Conventions may differ between programming languages.

If software is distributed as a plain archive (tar.gz, tar.bz2, zip or otherwise), it must create a single directory when unpacked rather than spread its contents over the current working directory.

IS6 – Is the software properly structured when installed?

When the software is installed, it should adhere to file placement standards set by the target platform. On Unix-like operating systems (excluding Mac OS X), the Filesystem Hierarchy Standard[3] (FHS) should be followed.

[3] `http://refspecs.linuxfoundation.org/fhs.shtml`

IS7 – The software must include a README.

The `README` file must contain links to the project website, as well as license and copyright information. The README should be either plain-text or preferably in an unobtrusive mark-up format such as Markdown[4] or ReStructuredText[5].

If a public version control platform such as GitHub is used (see AC2), the README will usually be visualised on the repository page.

IS8 – Are there facilities to uninstall the software?

This is usually trivial when the software complies to IS4 and/or IS3, or if the software installation is self-contained in a single directory. In all other instances, proper uninstallation facilities should be explicitly implemented.

IS9 – Are the system requirements such as target platform clearly advertised?

The project website must state clearly state what target platform(s) the software is expected to run on. System requirements in terms of computing resources must be stated if the software employs more than insignificant resources (see PF2). (related also to B3 and IS2)

5.6 Performance

The guidelines in this category are typically hard to assess, as they require an intricate knowledge of the design of the system.

PF1 – Does the software perform its function(s) efficiently?

Software should employ efficient algorithms to perform its task. This is quite non-trivial to assess, especially from a third party perspective, but obvious performance bottlenecks *may* be an indication of sub-optimal design choices.

(Consider also U4)

PF2 – Does the software make a reasonable demand on computing resources?

Software should not place an unreasonable demand on computing resource (memory, CPU time). Overuse of resources *may* be an indication of sub-optimal design choices. This is not easy to assess, but if the software does make high demand on particular resources, then this should be clearly advertised and explained (in the documentation for instance) and also stated as part of the system requirements.

Be aware that there is often a trade-off between speed and memory usage. The developer of the decision must make an appropriate decision favouring one or the other given the task at hand.

[4] `http://daringfireball.net/projects/markdown/`
[5] `http://docutils.sourceforge.net/rst.html`

PF3 – Does the software make efficient use of available resources?

Software should make use of available resources if it helps getting the job done faster. For instance, if multiple CPU cores are available and the task at hand would benefit significantly from parallelisation, then the implementation should be multithreaded to make use of said cores.

PF4 – Is the interface responsive?

GUIs and WUIs should be responsive and deliver clear feedback when the user is to await the completion of the task. GUIs and WUIs should use asynchronous methods for handling user-interface interaction. Interaction with the interface should not be blocked needlessly.

Webservice interfaces need to be similarly responsive and not time out. Server interfaces furthermore demand multi-threading and need to be able to handle multiple clients concurrently.

PF5 – Does the software scale as intended?

Proper facilities should be implemented to allow scaling *if* the software is intended to handle big data and/or large user bases. Performance should remain acceptable at such levels. Scaling can be horizontal, where multiple CPU cores or multiple machines are put to work to distribute the load, or vertically by simply adding more resources. In a data-sense, horizontal scaling often implies a partitioning of the data.

6 Quality Assessment Criteria - Sustainability and Maintainability

6.1 Identity

ID1 – Does the software provide a clear and unique identity?

Software should be identifiable by a clear name that does not clash with others in its application domain and in the wider generic software domain.

ID2 – Does the software have a website?

The software should have a website that describes it, allows users to obtain it as well as its documentation. The website should be a portal to everything related to the software.

It is recommended to have a dedicated domain name or subdomain.

ID3 – Does the project name not violate an existing trade-mark?

The project name should not infringe on existing trade-marks.

6.2 Copyright & Licensing

CP1 – Has an appropriate open-source license been adopted?

Software should be released as open source under an appropriate license. The license should be recognised by the Open Source Initiative[6] or Free Software Foundation[7]. Open source licenses generally fall somewhere on a spectrum between the following two flavours:

- Copyleft license (GNU Public License (GPL) and variants): If the code is modified and distributed, the modified code must be distributed under the same license. Also, if your code is licensed in this way and is used as a library or imported module by other software, then that software too is considered a derivate work and must be distributed under the same license. This ensures that anybody who derives a work from your code is obliged to distribute it as open source (free[8] software) as well.
- Permissive license (MIT, Apache License, BSD license): Allows modifications and reuse with fewer restrictions. Generally anybody is free to use the code and use it in closed-source (non-free) software.

The decision which license is most appropriate should be made on an individual basis. Note that none of the major licenses prohibits commercial use, nor is it recommended to do so.

The license chosen for the software may never violate the licenses of any of the software's dependencies, if there are linking clauses in those. [9].

Any form of closed-source software is strongly discouraged as it is found to be at odds with a scientific method that relies on methodological transparency, reproducibility and peer review. The CLARIAH project is explicitly committed to open source and "aims to create an inclusive open-source engineering community that will carry on providing new tools and support for users after the end of the CLARIAH project" [Filarski, 2015].

CP2 – Is it clear who wrote the software, who owns the copyright, and what the license is?

Authors, copyright and license should be clearly mentioned on the project website, as well as in the source code. We strongly recommend each source file to contain a comment header stating this information.

CP3 – Are the funders acknowledged?

Any external funders of the software should be publicly acknowledged on the project website, and in the `README`.

[6] https://opensource.org
[7] http://fsf.org
[8] Free as in free speech, not free beer.
[9] This implies you that if you use GPL libraries, you can no longer release your software under a more permissive license such as MIT.

6.3 Accessibility

AC1 – Is the source code maintained in a version control system?

Source code must always be kept under version control. This enables developers to collaborate, maintains a perfect version history, and allows everyone keep a track of changes. Absence of version control is bad development practice and unsustainable.

AC2 – Is the source code in a public version-controlled repository?

The version control repository should be public (read only) in the spirit of both open source, as well as the scientific method (transparency, peer review, reproducibility).

It is recommended to use a large sustainable third-party platform (GitHub[10], GitLab[11], BitBucket[12], Launchpad[13]), as it offers a social dimension and is more likely to live beyond the lifetime of any current project funding line. Use of sourceforge.net, de-facto standard prior to the advent of current leader Github, is strongly discouraged as it is under ill management and no longer deemed secure. Public repositories tend to be free on these platforms and often come with many additional benefits, such as a good public issue tracker (see SP2).

We motivate our reasoning in favour of public version control repositories as follows:

1. increased visibility for the project, especially if a major external platform is used.
2. facilitates judgement of development activity of a software project.
3. facilitates collaboration, especially from external partners.
4. invites peer review
5. allows users to be more intimately aware of changes, which in an academic context may impact their experimental results.

(depends on AC1)

AC3 – Is there no restricted data in the public source code repository?

Privacy and security sensitive data (passwords, API keys), as well as other restricted data, should never be checked into public version control systems.

AC4 – Are there clearly marked formal releases of the software?

The software should be formally released when the developers deem it a good time. Each release should be clearly marked with the version number of the software, according to a consistent scheme. The version control system should also have clearly identifiable tags that mark the state of the repository at the time of this release.

[10] `https://github.com`
[11] `https://gitlab.com`
[12] `https://bitbucket.org`
[13] `https://launchpad.net`

AC5 – Is the software deposited in a public repository for the language?

The ecosystems surrounding various high-level programming languages offer public repositories in which installable software release packages should be deposited. These repositories function as a primary source for installation (see also IS3) and generally offer automatic dependency management.

Use of these repositories is strongly recommended practice if available.

- **Python** – Releases of Python software should be deposited in the Python Package Index: `https://pypi.python.org`
- **Perl** – Perl modules should be deposited in the Comprehensive Perl Archive Network (CPAN): `http://www.cpan.org`.
- **Java** – Java components built using Maven and various other build systems should be deposited in The Central Repository: `http://central.sonatype.org/`
- **R** – R packages should be deposited in the Comprehensive R Archive Network (CRAN): `https://cran.r-project.org`
- **Javascript** – Javascript software should be deposited in NPM: `https://www.npmjs.com`
- **Ruby** – Ruby software should be deposited to RubyGems: `https://rubygems.org`

AC6 – Is the software available in the target platform's software repository?

If the software has a well-defined target platform and the target platform has a public software repository available, then it is recommended to submit the packages to this public repository.

This may however conflict with AC5, be rendered obsolete by AC5, or render AC5 obsolete. The decision which form of installation takes precedence should be made on an individual basis, considering the nature and audience of the software.

- Linux/BSD/Unix – Distributions generally have their own official repositories. Package inclusion can be a lengthy process.
  - Arch Linux: Submit your package to the Arch User Repository[14] (AUR)
  - CentOS: Consult https://wiki.centos.org/HowTos/Packages/ContributeYourRPMs
  - Debian: Consult https://wiki.debian.org/Packaging
  - Fedora: Consult https://fedoraproject.org/wiki/Packaging:Guidelines
  - Ubuntu: Ubuntu inherits Debian's packages, therefore submitting Debian is recommended. Users can submit software to Package Archive (PPA).
- Mac OS X: use the Apple Store for traditional Mac applications intended for end-users; for the more Unix-style software, use a system such as Homebrew or MacPorts.
- Android: use the Google Store, fully open-source software may also be submitted to F-Droid[15]

[14] https://aur.archlinux.org
[15] https://f-droid.org

- iOS: Use the Apple Store
- Windows: (TODO)

It is not always time -and cost-effective to package and deposit the software, especially not when there are multiple target platforms. When the software is not intended for adoption by a wider audience, it can be considered to ignore this guideline.

AC7 – Is each software release deposited in a persistent store with a unique DOI?

Persistent stores aim to preserve research output. They aim for a longevity that can not be guaranteed by normal software repositories. Each software release should be deposited in a persistent store and receive a unique Digital Object Identifier (DOI), a persistent ID that is ubiquitous in the academic world. The DOI can in turn be referenced in citations from publications.

Users using Github (see AC2) and Zenodo[16] can set this up with little effort. [17]. Any releases made on GitHub will then automatically transfer to Zenodo and receive a DOI, no user intervention is necessary.

6.4 Community

This aspect concerns to what extent an active user community exists for the software.

CM1 – Is there evidence of the software being in use by others?

It is recommended to list on the project website if the software has users aside from the primary developers and their immediate institution, this may facilitate further adoption. It is also recommended to mention (on the project website) a citable publication that discusses software, and to actively recommend or even require others to cite this if they use the software.

A list of third-party publications citing the software is also recommended for the project page. Alternatively, it is lightly recommended to incorporate success stories or quotes from satisfied users on the project website.

In addition to the aforementioned, evidence for this activity may also be sought in contributions to the software by external developers (assuming compliance with AC2), or questions in the issue/bug tracker or mailing lists.

CM2 – Is there evidence of external developers?

External developers, not related to the original developers or their institution, are a good sign of community interest. The threshold for contributing is relatively high. Compliance with this indicator is greatly facilitated by compliance with AC2.

[16] `https://zenodo.org`, a research data repository.
[17] A guide for this is provided at https://guides.github.com/activities/citable-code/.

CM3 – Are there statistics available on software use?

To gain an impression of user interest, it is recommended to have statistics available on how often the software is downloaded or how often the project page is visited. Public availability of these statistics (stripped of any privacy sensitive information), is recommended.

6.5 Testability

TS1 – Does the project have unit tests, is there sufficient coverage?

Software should have unit tests that automatically test individual units of the source code. They verify the data and logic flow by testing whether the output, given certain input, confirms to expectation.

It is important that the tests cover enough of the source code. This is not trivial to assess, but automated tools and platforms are available that can help in this assessment, such as `https://coveralls.io` and `https://codacy.com`.

TS2 – Does the project have integration tests, is there sufficient coverage?

Software should have integration tests that combine invididual parts of modules and see how they function as a group.

TS3 – Does the project have automated GUI tests?

If the software offers a non-trivial GUI or WUI, it should have automated tests that verify whether the interface functions as intended.

TS4 – Are tests run automatically?

It is recommended that tests be run automatically, either through continuous integration testing or periodically at a predefined interval. Free public continuous integrations platforms such as Travis-CI[18], Gitlab-CI[19], Jenkins[20] can be hooked into version control systems (see AC1, AC2) with minimal effort, resulting in an automatic run of the test suite upon each commit and notifying the developer when the test suite fails.

It is strongly recommended that test results are publicly available, so users can more quickly assess software quality.

6.6 Portability

Portability concerns the extent to which software can be used on multiple platforms.

[18] https://travis-ci.org
[19] https://gitlab.com
[20] https://jenkins-ci.org

PB1 – Is it clear for what platforms the software is written?

It should be clear, from at least the project website, for what platforms the software is intended.

PB2 – Is the software portable for multiple platforms?

It is recommended to support a wide variety of platforms rather than a single one. This, however, is not always feasible or cost and time effective.

PB3 – Does the software work on multiple browsers?

This concerns only web-based software with a significant client-side component. Such software should function under recent versions of all major browsers (Mozilla Firefox, Google Chrome, Internet Explorer / Edge, Safari, Opera), and never be limited to just one.

Moreover, it is recommended that such software does not rely on browser plugins that are themselves not portable. Adobe Flash or Microsoft Silverlight are two examples of badly portable legacy technologies that should be avoided, always use modern substitutes (HTML5) instead.

6.7 Supportability

To what extent will the product be supported currently and in the future?

SP1 – Is it clear whom to contact for support?

It should be clear where to go for support. A project must have a contact e-mail address. If an issue tracker is present (SP2), it should be clearly advertised as well.

SP2 – Are there public support channels available?

A public issue/bug tracker is strongly recommended. It allows everyone to post bugs or features requests and allows users to see what issues are current, how they are resolved, and if they are resolved in a timely fashion. It also prevents duplication of issues, and gives a platform for tracking feature requests.

Alternatively, a mailing list may as a lesser substitute for an issue tracker. It can also serve as an extra line of communications between users and developers. The mailing list must allow anyone to subscribe and must have a public archive allowing users to follow any issues and follow development.

Other extra support channels may take the form of an IRC channel or services such as Slack[21] or Gitter[22].

[21] `https://slack.com/`
[22] `https://gitter.com`

6.8 Analysability

This section concerns the extent to which the source code can be understood. At this level, the source code is inspected in closer detail.

AN1 – Is the source code structured adequately?

Source code should be modular, i.e. it should be structured into multiple modules/packages, following the requirements and conventions of the programming language.

The structure of the source code should bear a clear relationship to the architecture or design of the software.

(See also R3)

AN2 – Is the source code commented adequately?

The source code should contain comments explaining what major blocks do.

AN3 – Do the comments generate API documentation?

The comments use a mark-up that allows them to be used directly as the source for the generation of the API reference documentation. This is accomplished using document generation tools such as Doxygen[23], Sphinx[24] or Javadoc[25].

AN4 – Is the source code cleanly laid out?

The source code should follow a proper indentation convention.

AN5 – Are sensible names used?

Do the classes, functions and variables in the source code use sensible names and do they follow a consistent naming scheme that is conventional for the programming language?

AN6 – Are there no (large) blocks of commented out code or obsolete files?

There should be no large blocks of commented-out code, nor obsolete or alternate versions of files that bypass proper version control. (this depends on compliance with AC1)

AN7 – Are all TODO comments resolved?

There should be no important TODO comments, if there are they should at least be clearly described in the issue tracker (SP2).

[23] `http://www.doxygen.org`
[24] `http://www.sphinx-doc.org`
[25] `http://www.oracle.com/technetwork/java/javase/documentation/index-jsp-135444.html`

AN8 – Does the project have recommended coding standards?

A project should have recommended coding standards to which contributors should adhere. These standards should be consistent with the larger community of generic coding standards for the programming language.

6.9 Changeability

CH1 – Is the project open to contributions from third parties?

A project is recommended to be open to outside contributions. Community involvement is a major factor in the success of a software project.

This depends on AC1, AC2, and facilitates COM1 and COM2.

CH2 – Does the project have guidelines for contributions?

If contributions are desired, a project is recommended to have guidelines for contributors. These must be publicly available. See also AN8.

CH3 – Are all source code changes, and their authorship, publicly visible?

Collaboration requires awareness of what changes, when it changes, and who changes it. This depends on AC1, heavily facilitated by AC2.

Alternatively, commit messages can be forwarded to mailing lists, Slack, IRC chat, or whatever is deemed appropriate.

CH4 – Is the software sufficiently backward compatible?

Software should be backward compatible with old versions of itself with respect to public interfaces and data input. Backward compatibility changes and deprecation may occur but should always be clearly announced ahead of time.

CH5 – Is there a roadmap for future changes?

Software still under active development should have a roadmap. This may take shape either as an explicit roadmap or implicit in the issue tracker (SP2) through the assignment of milestones.

CH6 – Does the website mention how the software is funded and when funding ends?

Funders should be acknowledged publicly and people should be aware when software is no longer actively developed.

6.10 Reusability

R1 – Does the software offer all the appropriate interfaces?

The use of multiple interfaces enables reusability. Different users should be able to access the software at different levels. End-users usually require a GUI or WUI, developer-users prefer a command line interface (CLI) and/or webservice interface, developers needs a software library with API.

Assessment of this depends greatly on the intended audience of the software (see U2) and desired level of reusability.

R2 – Is the software modular, can the software support multiple interfaces?

Software should be set up in such a manner that higher-level interfaces can be constructed on its lower-level components. This implies that there should be a clear separation between front-end and back-end. In order to achieve this it we strongly recommend software to be set up in a modular fashion, allowing reuse of its components without the need to modify these components.

Modularity can be expressed as layers, from low-level to high-level:

- Classes and functions are defined at the source code level. (see R3)
- Libraries group and expose these publicly, described by APIs.
- Command Line Interfaces use the libraries.
- Servers/daemons use the libraries, networked clients use the server/daemons.
- GUIs and WUIs use either the libraries, the CLI tools, or act as a networked client to the servers/daemons.

Whenever two or more of these layers are intrinsically merged, reusability potential is lost. For instance, monolithic software that offers only a GUI interface can not be readily adapted to add a CLI or web interface.

Maximum reusability is not always desired or time- and cost-effective. The desired degree of modularity and reusability is to be assessed on an individual basis.

R3 – Is the software's source code set up in a modular fashion?

The software's source code should define clearly delimited classes and functions, following the paradigm of the programming language.

(See also AN1)

6.11 Security & Privacy

SC1 – Is the software free of obvious security flaws?

Software should be secure and have no holes that allow unauthorised users to gain access. Developers should take care to avoid common attack vectors such as shell injection, SQL injection, cross-site request forgery, buffer overflow. Proper validation of user input is a major factor in preventing security holes

Assessment of security is an art in itself and non-trivial.

SC2 – Is user privacy secured effectively?

Privacy-sensitive user data must be treated with care. Passwords must never be stored in unhashed form, private keys must never be shared. Any compromises to privacy must be clearly laid out in a privacy policy. Strongly related to SC1. Also see AC3.

6.12 Interoperability

IP1 – Does the software use appropriate open standards for data?

Software should adhere to appropriate open standards as much as possible, i.e. it should be able to read input files and write output files in open standards. Support of multiple open standards is recommended. Conversion may also be mediated through other third party tools. A counter indication for this is when software uses its own ad hoc format when decent open alternatives already exist.

6.13 Interoperability for CLARIAH

TODO
What specific interoperabilities does CLARIAH demand from software for integration into the larger infrastructure? External input is greatly appreciated here!

6.14 Governance

TODO
What are quality indicators for how software projects are run and managed? This is more on a high-level organizational level.

7 Quality Measurement

To achieve a measure of quality, evaluators can compute a quality score (Q, expressed as a percentage) per criterion, by counting how many of the indicators are answered with "yes" (p), and how many with "no" (n), following the simple formula in Equation 1.

$$Q = (p/(p+n)) \cdot 100 \tag{1}$$

A passing threshold can be defined, either on a per-criterion basis or just by applying the same threshold for all.

TODO
What quality thresholds do we set for CLARIAH, or is it up to institutions or even projects themselves? Do we need pre-set thresholds at all?

8 Implementation

TODO
How will these guidelines be implemented, used and reviewed? We are currently taking a closer look at the guidelines and looking for input from people in CLARIAH and beyond.The connection between data and software has to be worked out still. The criteria and indicators will eventually be published as an interactive survey as well. The idea is that developers can self-assess their projects, and adopters of software can assess whether software is of sufficient quality for their usage.

IMPORTANT NOTICE!
Request for Comment: In this first stage, this very first draft of the guidelines is being shared with people throughout CLARIAH with the request for initial feedback. • Do these guidelines seem helpful in the assessment of software quality and sustainability? Does it fill a need? • Are there specific criteria/indicators missing? Is there something more substantial missing? • Are there specific criteria/indicators you disagree with? • Which criteria/indicators would you consider particularly important? Which not? • Do you have input on any of the TODO items? • Do you have suggestions for improvement? Please send your feedback to proycon@anaproy.nl, reinier.de.valk@dans.knaw.nl and andrea.scharnhorst@dans.knaw.nl

References

S. Crouch, N. C. Hong, S. Hettrick, M. Jackson, A. Pawlik, S. Sufi, L. Carr, D. De Roure, C. Goble, and M. Parsons. The software sustainability institute: Changing research software attitudes and practices. *Computing in Science Engineering*, 15(6):74–80, Nov 2013. ISSN 1521-9615. doi: 10.1109/MCSE.2013.133.

Peter Doorn, Patrick Aerts, and Scott Luscher. Research software at the heart of discovery. Technical report, DANS, NLeSC, 2016. URL `https://www.esciencecenter.nl/pdf/Software_Sustainability_DANS_NLeSC_2016.pdf`.

Gertjan Filarski. CLARIAH Technical Plan v1. Technical report, CLARIAH, 2015. URL http://www.clariah.nl/files/wp/WP2_CLARIAH_Technical_Plan.pdf.

ISO/IEC. *ISO/IEC 9126. Software engineering – Product quality*. ISO/IEC, 2001. URL http://www.iso.org/iso/catalogue_detail.htm?csnumber=22749.

M. Jackson, S. Crouch, and R. Baxter. Software Evaluation: Criteria-based Assessment. Technical report, Software Sustainability Institute, 2011. URL http://software.ac.uk/sites/default/files/SSI-SoftwareEvaluationCriteria.pdf.

Appendix A: Checklist

ID	Indicator	Y	N	n/a
5.1 Understandability				
U1	Is it clear what the software does?			
U2	Is it clear for whom the software is intended?			
U3	Is it clear how the software works?			
U4	Is the software motivated?			
5.2 Documentation				
D1	Is there documentation?			
D2	Is the documentation accessible?			
D3	Is the documentation clear?			
D4	Is the documentation complete?			
D5	Is the documentation accurate?			
D6	Does the documentation provide a high-level overview of the software?			
D7	Are all the necessary audiences addressed, at their appropriate levels?			
D8	Does the documentation make use of adequate examples?			
D9	Is there adequate troubleshooting information?			
D10	Is the documentation available from the project website?			
D11	Is the documentation under version control?			
5.3 Learnability				
L1	Is there a *Getting started* guide?			
L2	Are there instructions for basic use cases?			
L3	Does the interface provide a help reference?			
L4	Is there API documentation for developers?			
L5	If the software is configurable, are the configuration options clearly explained?			

Table 2: Quality Assessment Criteria - Usability (1)

ID	Indicator	Y	N	n/a
5.4 Buildability				
B1	Are there good instructions for building/compiling the software?			
B2	Is an established automated build system used?			
B3	Are all dependencies listed and available?			
B4	Are there tests to verify the build has succeeded?			
5.5 Installability				
IS1	Are there easily accessible installation instructions?			
IS2	Are all dependencies listed and available?			
IS3	Are programming language's best installation practices followed?			
IS4	Is the software packaged according to standards for the target platform?			
IS5	Is the software package properly structured?			
IS6	Is the software properly structured when installed?			
IS7	The software must include a README.			
IS8	Are there facilities to uninstall the software?			
IS9	Are the system requirements such as target platform clearly advertised?			
5.6 Performance				
PF1	Does the software perform its function(s) efficiently?			
PF2	Does the software make a reasonable demand on computing resources?			
PF3	Does the software make efficient use of available resources?			
PF4	Is the interface responsive?			
PF5	Does the software scale as intended?			

Table 3: Quality Assessment Criteria - Usability (2)

ID	Indicator	Y	N	n/a
6.1 Identity				
ID1	Does the software provide a clear and unique identity?			
ID2	Does the software have a website?			
ID3	Does the project name not violate an existing trademark?			
6.2 Copyright & Licensing				
CP1	Has an appropriate open-source license been adopted?			
CP2	Is it clear who wrote the software, who owns the copyright, and what the license is?			
CP3	Are the funders acknowledged?			
6.3 Accessibility				
AC1	Is the source code maintained in a version control system?			
AC2	Is the source code in a public version-controlled repository?			
AC3	Is there no restricted data in the public source code repository?			
AC4	Are there clearly marked formal releases of the software?			
AC5	Is the software deposited in a public repository for the language?			
AC6	Is the software available in the target platform's software repository?			
AC7	Is each software release deposited in a persistent store with a unique DOI?			
6.4 Community				
CM1	Is there evidence of the software being in use by others?			
CM2	Is there evidence of external developers?			
CM3	Are there statistics available on software use?			

Table 4: Quality Assessment Criteria - Sustainability and Maintainability (1)

ID	Indicator	Y	N	n/a
6.5 Testability				
TS1	Does the project have unit tests, is there sufficient coverage?			
TS2	Does the project have integration tests, is there sufficient coverage?			
TS3	Does the project have automated GUI tests?			
TS4	Are tests run automatically?			
6.6 Portability				
PB1	Is it clear for what platforms the software is written?			
PB2	Is the software portable for multiple platforms?			
PB3	Does the software work on multiple browsers?			
6.7 Supportability				
SP1	Is it clear whom to contact for support?			
SP2	Are there public support channels available?			
6.8 Analysability				
AN1	Is the source code structured adequately?			
AN2	Is the source code commented adequately?			
AN3	Do the comments generate API documentation?			
AN4	Is the source code cleanly laid out?			
AN5	Are sensible names used?			
AN6	Are there no (large) blocks of commented out code or obsolete files?			
AN7	Are all TODO comments resolved?			
AN8	Does the project have recommended coding standards?			

Table 5: Quality Assessment Criteria - Sustainability and Maintainability (2)

ID	Indicator	Y	N	n/a
6.9 Changeability				
CH1	Is the project open to contributions from third parties?			
CH2	Does the project have guidelines for contributions?			
CH3	Are all source code changes, and their authorship, publicly visible?			
CH4	Is the software sufficiently backward compatible?			
CH5	Is there a roadmap for future changes?			
CH6	Does the website mention how the software is funded and when funding ends?			
6.10 Reusability				
R1	Does the software offer all the appropriate interfaces?			
R2	Is the software modular, can the software support multiple interfaces?			
R3	Is the software's source code set up in a modular fashion?			
6.11 Security & Privacy				
SC1	Is the software free of obvious security flaws?			
SC2	Is user privacy secured effectively?			
6.12 Interoperability				
IP1	Does the software use appropriate open standards for data?			
6.13 Interoperability for CLARIAH				
	[TODO]			
6.14 Governance				
	[TODO]			

Table 6: Quality Assessment Criteria - Sustainability and Maintainability (3)