

SUB RDD Technical Reference

8. Mai 2020

`<https://github.com/subugoe/rdd-technical-reference>`

Table of Contents

1	About This Document	4
2	Explanatory Notes	4
3	Style Guides	4
3.1	General	4
3.2	For Specific Programming Languages	4
4	Documentation	5
4.1	Doc Sprints	5
4.2	General Notions about Documentation	5
4.3	Developer Documentation	6
4.3.1	Software Architecture	6
4.3.2	API Documentation	7
4.3.3	CESSDA's Software Maturity Levels in RDD (CA1.3)	7
4.4	Admin Documentation	7
4.4.1	CESSDA's Software Maturity Levels in RDD (CA1.2)	7
4.5	Server Documentation	8
4.6	User Documentation	8
4.6.1	CESSDA's Software Maturity Levels in RDD (CA1.1)	8
5	Version Control	9
6	Bug Tracking	10
7	Software Tests	10
8	Building Code and Continuous Integration	10
8.1	Building Code	10
8.2	Packaging	11
8.2.1	CESSDA's Software Maturity Levels in RDD (CA5)	11
8.3	Continuous Integration	12
8.3.1	Sample Configuration of the GitLab Runner	12
8.3.2	Sample Configuration of the Jenkins CI (Multibranch Pipelines)	12
9	Deployment and Maintenance	13
9.1	Puppet	13
9.1.1	Monitoring	13
9.1.2	Release Management	13

10 Code quality level for RDD	13
10.1 Code Reviews	13
10.1.1 Sample Workflow: Code Reviews in SADE	14
10.2 Proof of Concept	14
11 Intellectual Property	14
11.1 Licensing	14
11.2 CESSDA's Software Maturity Levels in RDD (CA2)	15
12 CESSDA – Decisions	16
12.1 CA3: Extensibility	16
12.2 CA4: Modularity	16
12.3 CA6: Portability	17
12.4 CA7: Standards Compliance	17
12.5 CA8: Support	17
12.6 CA9: Verification and Testing	18
12.7 CA10: Security	18
12.8 CA11: Internationalisation and Localisation	18
12.9 CA12: Authentication and Authorisation	18
13 Retirement of Software	19
14 Helpful Links and References	19

1 About This Document

Author: Software Quality Working Group

Audience: Developers of the Research and Development Department of the Göttingen State and University Library.

Purpose: This guideline should help you getting started a new software development project (or improving an existing one!) in the Research and Development Department of the Göttingen State and University Library.

Our goal is to establish better software quality by following standards the developer team has mutually agreed upon. Roughly basing on the [EURL-SE Network Technical Reference](#), these standards are discussed, worked out, and decided in the Software Quality Working Group, which meets biweekly on Tuesdays at 13:00–14:00. However, they aren’t cast in stone, so in case you have a good idea for a better standard, feel free to contribute!

Status: This document is a living document and will be extended as soon as the Software Quality Working Group has agreed upon a new standard for software projects in RDD. TODOs and addenda of this document are maintained [here](#).

2 Explanatory Notes

CESSDA’s Software Maturity Levels (SML): Several organizations already have put a lot of work into developing metrics for good software. Since we didn’t want to reinvent the wheel, we decided to adapt (and modify if need be) a metric which suits our needs. [CESSDA](#) is one of the ERICs, and although it focuses on Social Sciences it has similar requirements regarding its software.

Throughout the document you will find sections with the heading “CESSDA’s Software Maturity Levels in RDD” in which we describe which of the SMLs we aim for and how we want to implement it.

3 Style Guides

3.1 General

The basic definitions are given by our [EditorConfig](#) file, `.editorconfig`, i.e. Unix line breaks and 2 space indentation.

3.2 For Specific Programming Languages

For the more prominent programming languages we have formatting and general style guides we ask you to follow:

- **Java:** The Java style guide can be found [here](#). It's based on the [Google style guide for Java](#) with some minor RDD specific settings. You can configure Eclipse to use it automatically at *Eclipse > Preferences > Java > Code Style > Formatter*. Just load the [RDD Eclipse Java Google Style](#) in the formatter preferences and use it in your RDD projects.
- **JavaScript:** For JS we use the [Airbnb JavaScript Style Guide](#).
- **HTML/CSS:** For HTML/CSS we agreed upon the [Google HTML/CSS Style Guide](#).
- **XQuery:** We use the [xqdoc style guide](#) with the following addenda:
 - use double quotes instead of single quotes (for easy escaping)
 - use four spaces for a TAB (because eXide switching the preferences in eXide's setting isn't permanent)
- **XSLT:** Since there is no official style guide for XSLT, we decided to write [our own](#), resulting from common best practices and own experiences within the department.
- **Python:** For Python [PEP 8](#) should be used, Django has a style guide based on PEP-8 with some exceptions: [Django-Styleguide](#). There are linters and tools like [flake-8](#) and [pep-8](#) available as support.
- **SPARQL:** For SPARQL there is not really any official style guide and there is no possibility to simply include any code style automatically using a code style file. We just collect some advice how to format and use SPARQL code [here](#).

4 Documentation

4.1 Doc Sprints

To ensure the best documentation of our code we meet on a weekly base for a code sprint to document everything we have coded throughout the week and haven't been able to document properly yet. The meeting takes place in the meeting room at 1pm. Cookies may be provided.

4.2 General Notions about Documentation

- Don't document a language's specifics, e.g. operators.
Example: Use of `=>` in the XQuery expression `replace("qbc", "q", "b") => substring(1, 2)` MUST NOT be explained in a comment.

- It is best to use a language's structure to document.
- Write the best documentation you can.
- Documentation and variable language is American English.
- Docs should be as close to the code as possible.
This refers both to the documentation in a repository and in the code itself, e.g. inline documentation.
- Every code repository MUST have:
 - a README.md file according to [this template](#) that contains
 - * link to original repository (if the software is forked or otherwise based on preexisting software)
 - * short introduction on what the repository is about
 - * a guide how to get the software running (if makes sense)
 - * link to demo instance
 - * example or demo installation
 - * link to license file
 - * contribution guide
 - * link to style guide
 - * link to bug tracker/project management system
 - * known issues
 - * badges to CI status
 - a LICENSE file

4.3 Developer Documentation

4.3.1 Software Architecture

Each software project should be documented using an architecture diagram that helps understanding its basic functionality (even though using tools to generate diagrams such as UML class diagrams doesn't seem to be possible in every case).

Examples:

- [Generating call graphs in eXist-db](#)

Call diagrams can be useful to follow code and service calls and should be existing for every API method.

4.3.2 API Documentation

- The docs should comprise used parameters, author and @since annotations
- **Example for Java**
- Links to callers must not be listed in the documentation, because this info will be deprecated soon. It is strongly recommended to use call stacks of tools like Eclipse (Java) and/or Call Graph Module (SADE).
- Document REST-APIs using **openAPI** if possible. OpenAPI docs should be located at `/doc/api` on servers.

4.3.3 CESSDA's Software Maturity Levels in RDD (CA1.3)

MUST MUST be SML2, which is defined as follows:

There is external documentation that describes public API functionality and is sufficient to be used by an experienced developer. If available, source code is consistently and clearly commented. Source code naming conventions are adhered to and consistent.

Actions to Be Taken in RDD

- provide a fully documented public API, e.g. by using OpenAPI
- naming conventions still have to be discussed -> style guide?
- reference to style guide used in the CONTRIBUTING/README file?

4.4 Admin Documentation

- The docs should comprise how to install the software, how to run and/or restart it, how to test the installation, ...

4.4.1 CESSDA's Software Maturity Levels in RDD (CA1.2)

MUST MUST be SML3, which is defined as follows:

There is a deployment and configuration manual that can guide an experienced operational user through deployment, management and configuration of the software. Exception and failure messages are explained, but descriptions of solutions are not available. Documentation is consistent with current version of the software.

Actions to Be Taken in RDD

- deployment: short deployment descriptions can be provided in the README, more detailed explanations should be kept separately (such as INSTALL(.md), linked from README)
- exception and failure messages are described in doc strings/function annotations
- function documentation should be generated automatically and made available/searchable in the web (such as readthedocs, javadoc html, etc. pp.)

4.5 Server Documentation

This type of documentation is provided and maintained in our DARIAH wiki space.

We have a template encompassing all information necessary: To create a wiki page for a new server navigate to the FE Server list, select “...” right beside the “Create” button and search for “FE-Server”.

4.6 User Documentation

This may encompass:

- how to use the software and APIs, FAQs, walkthroughs, ...
- guided tour (Bootstrap Tour) as user documentation
 - for SADE portal usage (such as Fontane, BdN, Architrave)
 - for complex Digital Editions
- screencasts

4.6.1 CESSDA’s Software Maturity Levels in RDD (CA1.1)

MUST MUST be SML2, which is defined as follows:

There is external documentation that is accessible and sufficient for an expert user to configure and use the software for the user’s individual needs. Terminology and methodology is not explained.

Actions to Be Taken in RDD

- a README(.md) has to be available in the source code repository (cf. [General Notions about Documentation](#))

SHOULD SHOULD be SML3, which is defined as follows:

There is a user manual that can guide a reasonably skilled user through use and customisation of the software to the user's individual requirements. Documentation is consistent with current version of the software.

Actions to Be Taken in RDD

- a more detailed explanation is available for the user at some place (such as user guide in wikis, etc. pp.)
- docs should also be provided in a `docs` directory in the source code repository
- docs are revised regularly during our doc sprints

5 Version Control

We exclusively use `git` in RDD. Please see <https://git-scm.com/doc> for information on how it works.

We recommend to use the [Git flow Workflow](#) (also consult the [Cheat Sheet](#)).

For using `git flow` it is safest to protect your `master` and `develop` branch on server side to avoid accidental pushes into these branches. All specific branches working on an issue described in a bug tracker may utilize the following naming scheme:

`[track]/#[ISSUENUMBER]-[KEYWORD]`

e.g. `bugfix/#12-flux-capacitor`.

A GitHub workflow used in DARIAH-DE and related services is described in the [DARIAH-DE Wiki](#).

You could also use the [GitLab flow](#) as a simpler alternative, which can be broken down into [11 Rules](#).

It is also recommended to automatically close issues via commit message; How this works exactly depends on the Git repository server. Issues can also be [referenced across repositories](#).

We use the following Git servers at the moment in RDD:

- Projects (GWDG) -> <https://projects.gwdg.de>
- GitLab (GWDG) -> <https://gitlab.gwdg.de>
- github.org -> <https://github.com/subugoe>

Which one is suitable for you depends on:

- the project you are working on
- existing code
- whether or not you want to use CI/CD or GitLab Runners
- ...

We have got an [RDD team on GitHub](#). Feel free to join us!

Consider mirroring of repos for project visibility (e.g. mirror GitLab/Projects code to GitHub?)

6 Bug Tracking

A bug tracking system is obligatory! Please use the respective bug tracking system of your repo and/or project management solution (please see chapter [Version Control](#))!

7 Software Tests

We aim to have a test coverage of **100%** (except for getter and setter methods). This is understood on a component level, which means that every method should have at least one test. Whether you achieve this by Test Driven Development (TDD) or not is specific to your preferred way to work.

Please keep in mind not only to write a test for each of your functions but also to consider all possible outcomes. It is e.g. not sufficient to test if a function creates a file if the written content depends on variables etc.

Examples for writing tests in different programming languages are:

- [XQuery](#)

8 Building Code and Continuous Integration

8.1 Building Code

Ideally, we use build tools to conveniently get a software running. The reason for using a build tool is to be able to build and/or test a code project with one command (after checking out). Another reason is to include dependency management.

Build tools we are using at the moment

- **bash scripting:** (bdnPrint, FontanePrint)
- **eXist:** Ant (SADE)
- **Java:**
 - Maven (TextGrid)
 - gradle (TextGrid)
- **JavaScript:**
 - bower (DARIAH-DE GeoBrowser, tgForms)
 - cake (tgForms)
 - NPM (DARIAH-DE Publikator, tgForms)
 - rake (DARIAH-DE GeoBrowser)
- **Python:**
 - make (Sphinx documentation)
 - PIP (DiscussData)
- **Ruby:** bundler (DARIAH status page)

8.2 Packaging

8.2.1 CESSDA's Software Maturity Levels in RDD (CA5)

MUST MUST be SML5, which is defined as follows:

Demonstrable usability: A Continuous Integration server job (or equivalent) is available to deploy the packaged/containerised software. Administrators are notified if deployment fails. Versions of deployed software can be upgraded/rolled back from a Continuous Integration server job (or equivalent). Data and/or index files can be restored from a Continuous Integration server job (or equivalent).

Actions to Be Taken in RDD

- examples for versions of deployed software: versioning of deb packages
- examples for rollback: rebuild index Elasticsearch from source data, restore database backup

8.3 Continuous Integration

We want to use CI as soon as possible in new projects. Please set up your GitLab project to show your pipeline-status and test-coverage for your default-branch under Settings/General->Badges.

The generic links for all projects are:

- Pipeline-status

- Link: https://gitlab.gwdg.de/{project_path}/commits/{default_branch}
- Badge image URL: https://gitlab.gwdg.de/{project_path}/badges/{default_branch}

- Test-coverage

- Link: https://gitlab.gwdg.de/{project_path}/commits/{default_branch}
- Badge image URL: https://gitlab.gwdg.de/{project_path}/badges/{default_branch}

The workflows we are using currently in Jenkins and GitLab Runner are:

- Code building
- Testing
- Code analyzer (Sonar)
- Packaging (JAR, WAR, DEB, XAR)
- Distribution (Nexus, APTLY repo, eXist repo)
- Release Management (via GitLab Environments and gitflow)

8.3.1 Sample Configuration of the GitLab Runner

There is a [full and documented example](#) of how the GitLab Runner is used in SADE.

8.3.2 Sample Configuration of the Jenkins CI (Multibranch Pipelines)

- On commit and push to the <https://projects.gwdg.de> gitolite repo (such as <https://projects.gwdg.de/projects/tg-crud/repository>) Jenkins on ci.de.dariah.eu is notified (see projects' gitolite configuration)
- Jenkins then does a checkout and build of configured branches (see Jenkins' project's multibranch pipeline configuration such as <https://ci.de.dariah.eu/jenkins/job/DARIAH-DE-CRUD-Services>).

- Stage *Preparation*: Prepare things
- Stage *Build*: Build, JAR, WAR, and DEB packages from source code, deploy JAR and WAR packages to the [Nexus repo](#). For further information on Nexus cf. [ci's server documentation](#). Jenkins is configured to deploy JARs and WARs via Maven and a Nexus deploy-account.
- Stage *Publish*: Publish DEB packages to the [DARIAH-DE Aptly Repo](#). Jenkins is using a shared library of scripts and publishing is divided into four conditionals: TG version, DH version, SNAPSHOT version, or RELEASE version due to given version suffixes!

9 Deployment and Maintenance

9.1 Puppet

For server configuration and setup we use puppet for most servers. The main puppet code is provided in GitLab <https://gitlab.gwdg.de/dariah-de-puppet>. The DARIAH-DE and TextGrid Repository module (dhrep) is contained in GitHub <https://github.com/DARIAH-DE/puppetmodule-dhrep>.

9.1.1 Monitoring

- Icinga probes for DARIAH-DE services <https://icinga.de.dariah.eu/icinga>
- Metrics for Sever specific monitoring <https://metrics.gwdg.de>

9.1.2 Release Management

10 Code quality level for RDD

10.1 Code Reviews

We want to ensure code review for all major commits, in gitflow for everything that is subject to be merged into `develop`.

For projects with more than one developer in the team it is preferred to have code reviews within the team, in other cases your friendly RDD developer team is on your side.

The main idea is to invite all developers to a MR, at least 2 approvals needed for MR taking place. This way everybody gets the chance to have a look at other people's code.

10.1.1 Sample Workflow: Code Reviews in SADE

1. A developer decides to work on a feature. She commits her changes to a separate feature branch. After some time she finishes the feature and wants it to be integrated in the development branch.
2. The developer creates a merge request and assigns everybody she sees fit to properly review her code.
3. To avoid diffusion of responsibility, she also assigns one of the chosen assignees as **MUST**. This means that this person has to approve the MR, otherwise the merge cannot be done. Although GitLab sends notifications to everybody who is newly assigned to a MR, she should notify the **MUST** assignee personally (in case he or she doesn't notice the mail sent by GitLab).
4. The **MUST** assignee reviews the changes according to style, variable naming, understandability, documentation provided, functionality, etc. If everything is to his or her liking, he or she approves the MR. The other assignees are free to review the code as well. **Note:** MRs without docs should not be accepted.
5. After the MR has been (dis)approved, the assignee removes his- or herself from the list of assignees. The **MUST** assignee informs the developer over the review being finished.
6. The developer merges her changes into the development branch.

10.2 Proof of Concept

When preparing a proof of concept that is always labeled **poc**, a code review is not necessary.

11 Intellectual Property

11.1 Licensing

- clarify software license before programming
- add license to code header

Best practice is to maintain a file listing all third-party packages that are part of the software. This list should hold the following metadata and **SHOULD** be prepared like the table below, always in alphanumeric order.

name	license	origin
foo	barware	github.com/foo/bar

Maybe the `license-maven` plugin will help you.

11.2 CESSDA's Software Maturity Levels in RDD (CA2)

MUST MUST be SML5, which is defined as follows:

There are multiple statements embedded into the software product describing unrestricted rights and any conditions for use, including commercial and non-commercial use, and the recommended citation. The list of developers is embedded in the source code of the product, in the documentation, and in the expression of the software upon execution. The intellectual property rights statements are expressed in legal language, machine-readable code, and in concise statements in language that can be understood by laypersons, such as a pre-written, recognisable license.

Actions to Be Taken in RDD

- see `rdd-technical-reference` for choosing the license
 - source code: use OSI approved licenses? (see <https://opensource.org/licenses>)
(++TODO discuss in `fe-develop++`) (how to chose a license?:
http://freesoftwaremagazine.com/articles/choosing_and_using_free_licenses_software_hardware)
 - * assets: CC0? CC-BY-SA? (++TODO discuss in `fe-develop++`)
- we reach SML5 by providing a license file on root level containing the license text (GPL wants its license text in a file called COPYING [see <https://www.gnu.org/licenses/gpl-howto.de.html>] other licenses use LICENSE)
- if the license text is not contained in the license file, we provide the full license text in another file (++TODO++) (<https://softdev4research.github.io/4OSS-lesson/03-use-license/index.html#add-a-licence-file-to-a-repository>)
- in the source code header the license statement is added to every file.
GPL example:

This file is part of FOOBAR.

FOOBAR is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

FOOBAR is distributed in the hope that it will be useful,

but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with FOOBAR. If not, see <<https://www.gnu.org/licenses/>>.

- the names of all contributed developers are stored in a contributors file on root level (every user that has committed in the repository)
- in the source code the contributors are added to each class/method/function/file/etcpp
- “and in the expression of the software upon execution“ – if applicable

12 CESSDA – Decisions

12.1 CA3: Extensibility

MUST be **SML3**:

„Use is possible by most users: Future extensibility is designed into the system for a moderate range of use cases. The procedures for extending the software are defined, whether by source code modification or through the provision of some type of extension functionality (e.g., callback hooks or scripting capabilities). Where source code modification is part of the extension plan, the software is well-structured, has a moderate to high level of cohesion, and has configuration elements clearly separated from logic and display elements.“

Actions to Be Taken in RDD: - Future extensibility is designed into the system - on RDD developer level - source code modification -> software is well-structured - provision of some type of extension functionality -> use of frameworks such as templating engines and localization frameworks - configuration elements clearly separated from logic and display elements - use SADE as example project

12.2 CA4: Modularity

MUST be **SML3**:

„Use is possible by most users: There is evidence that the architecture is open, with full structuring into individual components that provide functions or services to outside entities (i.e., open architecture); internal functions or services documented, but not consistently; modules have been created for generic functions, but modules have not been created for all of the specified functions; code within each module contains many independent logical paths.“

SHOULD be **SML5**:

„Demonstrable usability: It is evident that all functions and data are encapsulated into objects or accessible through web service interfaces. There is consistent error handling with meaningful messages and advice, and use of generic extensions to program languages for stronger type checking and compilation-time error checking. Services are available externally and code within each module contains few independent logical paths.“

Actions to Be Taken in RDD: - we NEED consistent error handling with meaningful messages and advice - if specific errors can occur -> create explicit errors - return and input types must be defined if language allows types - think of possible reuse of internal methods -> make them externally available - one module serves one purpose

12.3 CA6: Portability

(target platform: e.g. Docker container)

MUST BE SML5:

„Demonstrable usability: The software is completely portable to the target platform. In theory at least, the software will run on the target platform provided it is packaged/containerised.“

Actions to Be Taken in RDD: - examples for target platforms: Text-GridLab (Windows, Linux, Mac OS), Java Web Services (Web Application Server: Tomcat, etcpp), SADE (eXistDB, Linux, Windows), DiscussData-Django-App (jeder Host, für den Docker verfügbar ist)

++ Michelle möchte, dass wir mehr über Docker sprechen! ++

12.4 CA7: Standards Compliance

MUST BE SML3:

„Use is possible by most users: The software and software development process comply with open, recognised or proprietary standards, but there is incomplete verification of compliance. Compliance to recognised standards has been tested but this may not be for all components. There is documented evidence of standards being used, but not of the verification of components.“

Actions to Be Taken in RDD: - coding standards: code style, git (commit hooks), gitflow/gitlabflow, (semantic) versioning - software standards: documentation (JavaDoc, OpenAPI, etcpp), data and metadata formats, APIs (REST, SOAP, OAI-PMH, etcpp), license - CI standards: release workflow (?), deployment

12.5 CA8: Support

MUST BE SML1: **Actions to Be Taken in RDD:** - an organizational e-mail-address must be provided with the readme and in a convenient view, e.g. imprint/help/info etc.

SHOULD BE SML3: Actions to Be Taken in RDD: - provide support “near” the source code (discussable) - by any means, provide it in *one* location - regular and planned releases

12.6 CA9: Verification and Testing

Actions to Be Taken in RDD: - CESSDAs definitions seems a bit unclear to us - without referencing the SMLs we link our test chapter to CESSDAs document

12.7 CA10: Security

MUST BE SML2:

Actions to Be Taken in RDD: - every developer must have had a basic security training

SHOULD BE SML5:

Actions to Be Taken in RDD: - address security in every step of development (design, implementation, testing and verification, release)

TODO: - mit AL über Training für alle Developer sprechen - Anforderungen an Softwaresicherheit formulieren - Training erarbeiten (mit Security-Issue-Liste beginnen)

12.8 CA11: Internationalisation and Localisation

MUST BE SML1,5

Actions to Be Taken in RDD: - locale awareness is a high requirement for software with a monolingual target audience but you must provide it in english (and/or the target language) at least

SHOULD BE SML3

Actions to Be Taken in RDD: - if applicable, i18n and l10n frameworks should be used

nice to read: <https://bridge360blog.com/2011/11/25/software-design-considerations-for-internationalization-and-localization/>

12.9 CA12: Authentication and Authorisation

MUST BE SML2 SHOULD BE SML4

In this case we cannot give a general recommendation since the way authentication and authorisation is implemented inherently depends on the software’s functionality. Instead of developing an own solution rely on DARIAH’s AAI whenever possible.

Actions to Be Taken in RDD: - never share passwords - use Shibboleth whenever possible and reasonable

13 Retirement of Software

- clarify if software is no longer supported

14 Helpful Links and References

- EURISE-network technical-reference:
<https://github.com/eurise-network/technical-reference>
- DHTech – An international grass-roots community of Digital Humanities software engineers:
<https://dh-tech.github.io>
- The Software Sustainability Institute, Guidelines and Publications:
<https://www.software.ac.uk>
- The Joel Test: 12 Steps to Better Code:
<https://www.joelonsoftware.com/2000/08/09/the-joel-test-12-steps-to-better-code>
- Software Quality Guidelines:
<https://github.com/CLARIAH/software-quality-guidelines>
- Software Testing Levels:
<http://softwaretestingfundamentals.com/software-testing-levels>
- Netherlands eScience Center Guide
<https://guide.esciencecenter.nl>