Quality Committee Handbook

Release: 2.0.1

created by

Quality Committee

Revision History

Version	Date	Reviser	Description	Status
1.0	2016-06-20	M. Jung	Alignment in AC/QC conference call	Released
1.0.11	2016-08-12	M. Jung	Removed UML tool discussion, added UML tool decision in design documentation, Added configuration documentation, Removed stale “Related docs” appendix Removed “Meeting structure” appendix (structure already implemented). Added details for commit rules, library usage, and GUI styleguide application.	Draft for v1.1
1.1	2016-08-18	A. Goering	Alignment in AC/QC conference call	Released
1.1.1	2017-09-11	A. Goering	Added detailed package naming, cleaned up.	Released
2.0	2018-09-07	M. Berth, C. Dohle, M. Gründler, T. Meyer, E. Schlenker and others	Clarification and extension of chapters 2-4 and appendix	Draft
2.0.1	2018-10-15	E. Schlenker	Revision after translation and lectoring	Released

Formal

According to a decision of the Quality Committee, this document is written in english.

Document control:

Author: Dr. Martin Jung, martin.jung@develop-group.de (representative of the quality committee)
Reviewed by: SC, PPC, and AC of openKONSEQUENZ
Released by: QC

Released versions will be made available via the openKONSEQUENZ web site.

Related documents

Document	Description
BDEW Whitepaper	Whitepaper on requirements for secure control and telecommunication systems by the german BDEW Bundesverband der Energie und Wasserwirtschaft e.V. https://www.bdew.de/internet.nsf/id/232E01B4E0C52139C1257A5D00429968/$file/OE-BDEW-Whitepaper_Secure_Systems%20V1.1%202015.pdf
BSI-Standard-100-2	The IT-Grundschutz Methodology is a BSI methodology for effecitve management of the information security for adaption to the situation of a specific organization. https://www.bsi.bund.de/SharedDocs/Downloads/EN/BSI/Publications/BSIStandards/standard_100-2_e_pdf
ISO/IEC 25010	International Organization for Standardization ISO/IEC 25010:2011 Systems and software engineering -- Systems and software Quality https://www.iso.org/standard/35733.html
ISO/IEC 27000	International Organization for Standardization ISO/IEC 27000:2018 Information technology -- Security techniques -- Information security management systems -- Overview and vocabulary https://www.iso.org/standard/73906.html
oK-Charter	The openKONSEQUENZ charter https://wiki.eclipse.org/images/f/f5/20150623a_openKonsequenz_V14-3_%283%29.pdf
oK-GUI-Styleguide	Style guide for module developers of openKONSEQUENZ modules according to the graphical user interface. http://wiki.openkonsequenz.de
oK-Interface-Overview	AC handbook external but related AC document (appendix), where the Interfaces and the overall oK-CIM-Profile of oK-Modules are described in short as well as showing the Building Block View Level 1. (https://wiki.eclipse.org/OpenKONSEQUENZACQCRichtlinien)
oK-Module-Tender-Call	The openKONSEQUENZ project planning committee prepares a document which describes the requirements to the development for each module. With this document it calls for tenders at software developers (module individual)
oK-Module-Tender	The software developers answer to the oK-Module-Tender-Call (module & developer individual)

oK-Vision	The oK document “Vision/Mission/Roadmap” - it is currently not available online.
oK-Website	The website of openKONSEQUENZ www.openkonsequenz.de
Architecture Committee Handbook	Textural guideline for module developers of openKONSEQUENZ modules with respect to architectural design & documentation. https://wiki.eclipse.org/OpenKONSEQUENZACQCRichtlinien
oK-Wiki	The openKONSEQUENZ Wiki also contains the **glossary** of terms! http://wiki.openkonsequenz.de/Glossar

Table of Contents

1. Introduction

This document describes the role of the quality committee (QC) in openKONSEQUENZ. It defines quality criteria for the projects to be developed in the context of openKONSEQUENZ. While QC acts as a control authority, the Architecture Committee (AC) creates and maintains the overall software architecture and establishes rules for the creation, documentation and implementation of architecture in the various projects. The rules for architecture are described in the oK-AC-Handbook (in the following: AC-Handbook).

1.1 Mission of QC

The mission of the quality committee (QC) is to define, maintain, and enforce the guidelines for development in the openKONSEQUENZ group.
The QC defines the quality standards for project work (process quality) and project outcome (product quality). It provides procedure and methods for projects to use and selects tools to be used (e.g. for documentation).
The QC also defines the integration and QA environment (see chapter III below).
The QC also monitors and evaluates projects and gives feedback to the projects and to the AC, the project planning committee (PPC), and the steering committee (SC). The QC supports the other committees in their work.

1.2 Responsibility of QC

The quality of the software code base in openKONSEQUENZ is ultimately the responsibility of the QC. The AC will focus on the construction of the software and on facilitating the quality goals; checking and controlling them is the responsibility of the QC. Providing infrastructure for continuous tracking and reporting of the quality indicators during project work is also responsibility of the QC. If projects experience difficulties when applying the quality rules and regulations, it is the QC’s job to support the projects and to help find a viable solution for their work. Finally, the QC is responsible for analysing project outcome, and giving technically sound recommendations to the PPC and SC on whether to accept or to decline a project outcome.

1.3 Competence of QC

The QC may reject a project outcome based on quality analysis. If either the source code itself, the software running on the QA environment, the documentation, or any other mandatory project result does not meet the quality requirements, the QC will advise the SC and the PPC to reject the project’s outcome and to plan re-work. Such a recommendation must be made in writing and published via the openKONSEQUENZ mailing list. The recommendation may be challenged by the project, and decision will then be escalated to the SC.

Whenever project proposals are prepared, the QC may alter the description of the project scope to include key quality requirements and the core quality rules, before a public call for proposals. At minimum, the QC will ensure that this document is part of every official project proposal. Furthermore, the QC will only alter the PPC’s description of a project if specific quality aspects are especially important or have a high impact on the project scope.

1.4 Ownership of documents

To define the quality rules and to enforce them, the QC creates, maintains and releases documents, and defines the development environment. The list of documents are:

- This document “openKONSEQUENZ: Quality Committee Handbook”
- A directory of coding guidelines (currently co-located in this document, see *Appendix D*)
- A list of acceptance criteria for project outcomes
- A repository containing standardized test data (to be initialized)

The build infrastructure, its technical specification, and a HOWTO for its usage are also owned by the QC.

2. Project/Module Classification and Review

The items described in this chapter should be compiled or prepared by the Project Planning Committee (PPC) in preparation for a tender. The objectives defined by the PPC with regard to criticality, complexity and quality will become mandatory components of a tender.

The following sections describe how modules and projects are classified by the Project Planning Committee (PPC) based on their criticality and complexity and which review method applies according to the classification. In addition, these goals will be extended to include additional quality aspects of ISO/IEC 25010 with expected software quality features.

2.1 Project and module Classification

The project and its modules have to be classified using the following rules:

- Complexity on a scale of Small, Medium, Large
- Criticality on a scale of Normal, High, Very High

2.1.1 Criticality

This document classifies criticality as a vector in three dimensions, *availability*, *confidentiality*, and *integrity*. The Criticality values are related to the definitions in the International Standard ISO/IEC 27000. Whenever one dimensions is classified “high”, the project’s criticality is considered to be “high”, and whenever one dimensions is classified “very high”, the projects’s criticality is considered to be “very high”. The classification should be done according to the approach of Determining the protection requirements, described in the BSI-Standard 100-2.

2.1.2 Complexity

Complexity is related to the size of a module, and can be coupled to typical sizing like story points or (in this case) Person Days (PD).

2.1.3 Quality

Based on the defined IT protection needs and quality objectives, the project participants will define mandatory and, above all, testable specifications, similar to the technical Definition of Done (DoD) to be met for the project. In particular, during the review, we need to check for and ensure compliance with these specifications. The review criteria listed in Chapter 2.4.4, based on existing best practices from the oK projects, are helpful for this. The goal of this initial evaluation of quality criteria for the oK module under consideration is to establish a harmonised concept of the quality criteria expected of the software for the given oK module.

2.2 Review method

The overall classification have to be documented in each project proposal. It is required to determine the review methods as stated in the table below:

Complexity / Criticality	Small (< 120 PD)	Medium (120 - 240 PD)	Large (>240 PD)
Normal	No manual review required	Peer Review	Peer Review
High	Peer Review	Peer Review	Walkthrough
Very High	Peer review	Walkthrough	Deep inspection

The review methods apply to documentation, code, test specifications, and test protocols. "No manual review" means automated checks on the code are sufficient. "Peer review" means an offline check by AC or QC representatives. A representative is nominated by the committees and is not necessarily a member; e.g. he/she may be another committer or developer, preferably from another project. "Walkthrough" means an online check by an AC or QC representative, the author explains the solution. "Deep inspection" means an online check by an AC member and a QC member; the author and the project lead explain the solution. The author is responsible for organising the review.

2.3 Statement of Qualities

Each project shall document its quality requirements in the project's architecture documentation (according to the AC handbook). Typically, not all quality attributes are relevant for a project or module. The quality dimensions as defined in ISO 25010 are a frame of reference. Whenever a quality is of special concern to the project, the impact on the solution has to be documented. The AC refines system requirements down to project or module requirements. For example, if a persistence module impacts the overall availability of the system, the AC breaks down the overall system availability requirements to this module. The module has to document these requirements, and has to state how they are implemented, verified and validated.

Even before a tender, mandatory quality criteria as per ISO 25010 need to be evaluated and documented for each oK module in line with the specification by the PPC (e.g. LeadBuyer, ProductOwner). *Appendix A - Project Guidelines* provides a set of guidelines for assessing these quality criteria and are intended to help Project Planning with project specification, right from the earliest phases. The table included in this appendix serves as a template for an inspection catalogue which is required for all project participants.

To evaluate phases 4 and 5, we need to define required scenarios, to be used later on as a basis for review and approval. A scenario describes a specific situation for the given quality criterion, the desired behaviour and how this can be measured or tested and ensured.

For each oK module, at least one quality criteria evaluation must be performed, and the corresponding quality gates defined and saved in the designated project folder. The essential quality criteria for an oK module should be adjusted in accordance with the resulting architectural decisions and documented. In addition, the quality criteria for an oK module should always be defined with due consideration of the quality criteria for the entire oK platform.

2.4 Project Guideline

The *project procedures* defined in *Appendix B - Processes* serve as a basis and guideline for the implementation of a project. In order for all project participants to maintain transparent expectations throughout the project, the following process steps must be implemented in line with the project procedures coordinated in QC.

2.4.1 Project Planning

The PPC (e.g. LeadBuyer, ProductOwner) will evaluate and, to the extent possible, define the project evaluation ahead of time, before a tender:

- Determine project type
    - New development or
    - further development

- Define and describe protection needs categories & quality objectives
    - The table in *Appendix A* as per ISO 25010 serves as a guideline

The project evaluation will be a required component of a tender. The evaluation of the specifications established by the PPC (e.g. LeadBuyer, ProductOwner) is especially relevant for further development projects and should be coordinated in a workshop with all the project participants.

2.4.2 Project Initialisation

With openKONSEQUENZ, the development project and any review project that might be required are typically tendered and commissioned independently of each another. This way, the different project goals resulting from different commissionings cannot be excluded. To ensure transparent expectations and coordinate required review criteria, an initial project kick-off as per *Appendix B - Processes - Project initialisation* involving all project partners should be implemented to establish required, testable project goals.

As a result of different project preconditions, e.g. new or further development project, as well as different technologies and rapidly developing innovations, review criteria can only be harmonised conditionally. The QC manual constitutes a harmonised set of guidelines for implementing a software project in the context of openKONSEQUENZ, based on the quality criteria defined in the oK panels.

For new developments, the standards defined in the QC manual are mandatory. If any project-specific adjustments are beneficial and necessary, the PPC (e.g. LeadBuyer, ProductOwner) can request a joint project initialisation. The project-specific adjustments will be coordinated based on the standards and recommendations defined in the QC manual and documented as requirements in a coordinated oK development process. This might be the case, for example, if the planned oK module needs to meet high security standards.

For further development projects or donations, a corresponding workshop is obligatory. The sub-process *Defining and documenting the degree of fulfilment* stipulated in *Appendix B-Processes* and the sub-process *Project Coordination* serve as guidelines for project implementation. These specifications are used to define shared acceptance criteria:

- Finalising quality objectives
    - The table in *Appendix A* as per ISO 25010, to be compiled by the PPC (e.g. LeadBuyer, ProductOwner) as part of the tender, serves as a guideline.

- Defining project goals
    - Defining the TARGET status &, if applicable, ACTUAL status (for further developments only); for further development projects, a Status Quo must be determined.

- Establish required review criteria / rules
    - based on Chapter 3
    - Code quality ⇨ establish toolset, rule set, formatter, etc.
    - Design quality ⇨ coordinate / establish necessary architecture deviations from AC manual, as well as required review criteria
    - Product quality ⇨ establish acceptance criteria with regard to sprint and commit cycles

All specifications described in this QC manual are considered mandatory for project implementation and must be coordinated with all project participants and documented as early as possible if any deviations are required. Deviations from the oK specifications (e.g. Files System Conventions, Configuration Management,…) must be documented in AsciiDoc format:

(/documentation/variations.adoc)

2.4.3 Project Implementation

This process is used to coordinate any necessary changes in the course of the project. The *project implementation* sub-process defined in *Appendix B - Processes* serve as a guideline for the implementation of the project.

In line with development, at the start of the development project, the reviewer must compare the current status of the project to the established target criteria, determining and documenting the current degree of fulfilment. This evaluation is followed by another coordination meeting between the project participants (see sub-process *Project Coordination*). After completion of all designated sprints, evaluation of the degree of fulfilment and review of the jointly defined acceptance criteria, acceptance procedures can take place.

Deviations from established specifications and resulting changes within individual sprints must be made transparent to all project partners and reviewed and approved by the reviewer as early on as possible. Approval by QC is not required, but, in the case of best practice, changes should be presented to QC. Deviations from existing oK specifications (e.g. Files System Conventions, Configuration Management, …) must be documented in AsciiDoc format:

(/documentation/variations.adoc)

2.4.4 Project Acceptance

The *Ending* sub-process defined in *Appendix B - Processes* serves as a guideline for the completion of the project. This process is used for the publication of all project and review results as a basis for a subsequent evaluation by QC and, if necessary, updating of best practices from the completed project.

Acceptance takes place when, for all the acceptance criteria established at the beginning of the project, there is agreement on the degree of fulfilment, or identified deviations have been documented and accepted by all project participants.

3. Quality Rules and Guidelines

The following sections define the quality requirements on development in the openKONSEQUENZ group.

3.1 Code Quality

3.1.1 File System Conventions

Uniform appearance of the project directory structure and consistent naming of files are mandatory in collaborative work.

- The standard maven project directory structure[1] have to be used.
- The project shall provide AsciiDoc files for all documentation purposes. The files are enumerated below, see *Appendix C "Project Directory Layout"*.

If project-specific conditions make it necessary to deviate from the MAVEN standard project structure, this needs to be made transparent to all project participants and documented as early on as possible. For additional info, refer to the sections on best practices of the oK Wiki:

http://wiki.openkonsequenz.de/Hauptseite#File_system_conventions

3.1.2 Coding Guidelines

Uniform appearance of source code, the readability, and the avoidance of error-prone statements are key to good code quality.

- The coding guidelines have to be adhered to. See *Appendix D "Coding Guidelines"* below.
- Functions/Methods have to be documented using Javadoc, JSDoc, Doxygen, or equivalent source code markup systems.

The coding guidelines selected for openKONSEQUENZ will be implemented by rulesets of software tools (e.g. PMD, Checkstyle, FindBugs). They will then be enforced by the continuous integration procedure. The ruleset will be made available via the openKONSEQUENZ wiki or a source code repository.

For all oK projects, standardised coding guidelines are prepared as a rule set for the software tool SonarQube (https://www.sonarqube.org) and can be downloaded from the oK Wiki (http://wiki.openkonsequenz.de). These standardised coding guidelines are required for all projects and do not need to be individually coordinated for each project.

[1] https://maven.apache.org/guides/introduction/introduction-to-the-standard-directory-layout.html

In the oK Wiki, best practices are continually being added as experience from oK projects and as recommendations for new projects, e.g. how coding guidelines can be reviewed during the development period and in the continuous integration process.

It is also possible to integrate an oK project directly through the SonarQube instances of the Eclipse Foundation (https://sonar.eclipse.org/dashboard) and run a code analysis in conjunction with the sprint. Only standardised rule sets generated by the Eclipse Foundation and based on SonarQube can be accessed. Project-specific adjustments or changes to these standard rule sets are excluded so far.

If the standardised coding guidelines need to be adjusted, e.g. in case of further development projects, based on the standard rule sets, all the project participants collaborate at the beginning of the project to coordinate and define test metrics based on SonarQube standard rule sets that are specific but individually adapted to the technological requirements of the project. When checking the coding guidelines, refer to the *Defining and documenting the degree of fulfilment* process described in *Appendix B*. In addition, rule sets for each project can be individually expanded or adjusted. The project participants can also jointly agree on their own rule sets. If adjustments are necessary, a project-related SonarQube instance must be used. It must be ensured that if multiple rule sets are used, metrics do not overlap and different weightings are not assigned. For any necessary adjustments in the course of an ongoing project, the Project Implementation process described in *Appendix B* must be followed.

An oK module always uses a specified version of the rule sets. Deviations in the standard ruleset stipulated in the oK Wiki should always be versioned according to the project and documented. Ensure that this ruleset version is used in the continuous integration process. The source code documentation should be generated and packeted through the build system. Be sure to also select an appropriate formatter for the ruleset.

> *Note: The results of the tool assessment allow us to draw conclusions regarding the serviceability and indicate programme code sections containing indications of errors from certain error classes. An error can only be identified by manually reviewing these sections. Aside from assessing serviceability and checking for certain error classes, the static code analysis does not provide any evaluations of the remaining quality criteria as per ISO standard 25010. Due to complex safety standards, the SonarQube safety tests do not take the place of any safety tests. The "quality of the architecture" is not assessed. Here, too, it is only possible to identify indicators for weak points in the architecture, e.g. extreme complexity of module dependencies.*

3.1.3 Commit Rules

To guarantee an up-to-date view on the codebase, and to allow all developers to use functionality as soon as possible, it is important to commit early and often. Functionality that is presented in a Sprint Review Meeting has to be pushed to the repository before the Sprint Review Meeting. Smaller sets of functionality (e.g. an interface without implementation, an abstract class, etc.) should be pushed as soon as they are testable. To guarantee a continuous flow of development result, a push is required every two weeks (mid-sprint push & sprint review push).

If, for project-specific reasons, changes must be made to the release cycles or sprint times, e.g. due to dependencies from projects running in parallel, these must be made transparent to all project partners as early as possible and must be coordinated and documented no later than at the project kick-off.

3.1.4 Unit Tests & Code Coverage

Unit testing is a standardized method to assure quality at the source-code level.

- Unit tests have to be written and cover 80% branches.
- At least one unit test case must be written for each use case (or user story).
- Unit tests may use their own individual test data. If possible, they shall use standardized test data, in order to avoid “wrong” data.
- All unit tests have to be executed in the daily build of the integration environment.
- Coverage metrics have to be reported on a daily basis via the integration environment.

Based on the existing best practices of openK, it is necessary to achieve a test coverage of at least 80% (branches), with due consideration of the underlying SonarQube standard rule set and the associated quality gate. The reviewer must perform a corresponding evaluation in line with the *Project implementation* process (see *Appendix B - Processes*). In individual cases, project partners can agree to define certain test coverage deviations as proper, but these must be made transparent, coordinated and documented by Development as early as possible.

In addition, the code coverage must be determined, since this metric provides information on the completeness of software tests and the proportion of the test actually performed vs. the test scope that is theoretically possible. Depending on how we determine the number of theoretically possible tests, we differentiate between statement coverage, branch coverage, decision coverage and other degrees of coverage. For additional info, refer to the sections on best practices in the oK Wiki:

http://wiki.openkonsequenz.de/Hauptseite#Unit_Tests_.26_Code_Coverage

3.1.5 Configuration Management

The configuration (Versions of different components, of external libraries, of configuration files) have to be described in the maven POM files.

- All code, configuration files, data, and documents shall reside in openKONSEQUENZ’ git repositories.

Dependencies among components of the openKONSEQUENZ software and to/from external software components have to be explicit. This is achieved by maintaining maven pom.xml files. No local libraries (implicit dependencies) are permitted.

Dependencies on external libraries must be explicitly listed and specified. These must be stored in a technically processed format (e.g. xml or POM configuration) in the project folder in the following directory:

/dependencies

The data must include the complete name of the library, including the version. Indicating the version number is required. The specific versions must be indicated in order to prevent the version of a software library from being automatically assigned at the time the software artefact is created. This process should be assisted and harmonised through the use of a centralised library management system.

Preference should be given to approved libraries, based on best practices in the oK Wiki already in place through the IP check of the Eclipse Foundation and recommendations. Other libraries can be used, provided that this is justified and documented. However, this requires a justification of why the recommended library cannot be used in the project. No later than upon project completion, any libraries used which deviate from the recommendation list must be presented in the quality committee and added to the recommendation list in line with best practices.

If, based on oK-specific guidelines or experience from the openKONSEQUENZ development projects, libraries are identified which should not be used in oK modules due to certain reasons (e.g. safety problems, performance problems, ...), these libraries should be added to a central blacklist (with version number). The reason must be documented. These libraries should be integrated by the Quality Committee, similarly to the recommendation list. For Maven projects, the bill of material pattern (BOM) must be used; also refer to:

https://maven.apache.org/guides/introduction/introduction-to-dependency-mechanism.html

For Angular UIs, for example, the Maven “frontend-maven-plugin” can be used in order to produce a technical description of the required Angular tools.

To freeze versions for test and release, the versioning numbers shall conform to the pattern X.Y.Z.A, where:

- X: Major Release, incremented on major changes like interface definitions or end-user visible functionality
- Y: Minor Release, incremented on each delivery. Stable versions have even numbers, developer versions are odd numbers
- Z: developer tag, incremented on each code change
- A: Git-Hash, git-commit reference for easier handling

During development, feature branches have to be used; the name of the feature is added as part of the version string like this: X.Y.featurename.Z.A

All check-ins need to be commented. The comments have to be plain text and shall contain the following information:

- Topic (max. 50 characters)
- Task reference (to backlog item, requirement, defect, …; always using the ID of the task)
- Reason: Why was this change necessary?
- Rationale: What has been done (on an abstract level)?
- Side effects: What else has changed?

3.1.6 Build, Package & Test

Since the software has to be widely accessible, an automatic method of building, packaging and testing it is required. The maven tool will be used to control these tasks.

- The maven files shall encompass functionality to build the software, create runnable packages, run the unit-tests, and create all documentation files.

3.1.7 Diagnosis, Exceptions and Errors

Each module shall offer interfaces to access diagnosis information such as operational state, errors or exceptions. All errors and exceptions have to be logged to a plain text file. The format of log messages shall comply with the following rules:

- Exactly one line of text for each incident
- Each line shall start with a timestamp, a module designation, a thread ID in a comma separated line header: “YYYY-MM-DD_HH:SS:ssss, moduleXYZ, TID4711823, …”
- Other information relevant for the module may follow in a comma separated list
- Other information may contain json formatted object information. Json text is in one line and is correctly enclosed in curly brackets
- If human readable text is part of the log messages, it must be in English.
- Personalized Data and credential information must not be written to log files.

3.2 Design Quality

3.2.1 Design Documentation

All projects shall document their technical solution concept. The design documentation has to conform to the architectural guidance specified by the AC-handbook. Technical decisions have to be explained; their alternatives and consequences of the decision must be documented. UML description must be used as appropriate, according to the tooling decision, see Architecture Committee Handbook, Section 2.1.

All libraries used by a project have to be listed in the design document, together with a rationale for their usage. Libraries that are already cleared are default, they have to be used unless there are valid reasons against their usage (a list of cleared libraries can be obtained from the eclipse foundation). If a project chooses to override this default, it needs the approval of the architecture committee to do so. In that case, the project is responsible for doing the IP clearing. The clearing has to satisfy the publication license of openKONSEQUENZ. The clearing of the library has to be done as early and as fast as possible. If a clearing fails, the library must not be used; all development effort based on such a library is wasted.

3.2.2 Verification

One of the aspects of verification involves the static analysis of source code and the design and code reviews. These need to be performed according the review method matrix, see table "Review Method" above.

The other aspects are tests. Unit tests are required to check basic code functionality, and to maintain code quality. Design tests check the technical solution against the functional and qualitative requirements.

- Module test specifications have to be defined for each module. Using black box test specifications, the test specification specifies how the requirements and quality attributes are checked.
- Acceptance criteria have to be defined for the module test specification.
- The test data required to run the module tests have to be specified, with respect to the global data model and the global test data set.
- Mock functionality (https://en.wikipedia.org/wiki/Mock_object), required for the execution of the module test, have to be specified.
- Test specifications, test data, and mock functionality have to be formalized in test code that can be run automatically. This might not be possible for all test specifications. Manual test have to specified as a working recipe for a tester.

3.2.3 Design Reviews

Design Reviews have to be carried out according criticality and complexity rating and the resulting review method. Review records have to be stored in the filesystem according to Chapter V “Project Directory Layout”.

3.2.4 Document List

- Project architecture concept. Contains the mapping to overall architecture and the break-down into modules. Also, it contains the rationale for technical decisions and alternatives, the statement on qualities and the criticality/complexity rating. This document shall conform to the guidelines by the AC.
- How-to information for developers and integrators. Contains information on building, running, coding, and testing in this project
- Review records. Contains a list of participants, findings, and resolutions
- Test specifications for module integration test. Contains the strategy for integrating this projects modules, and a concept to test the integration.
- Test protocols. Contain the test setup used and a verdict. If appropriate, execution logs are also included.
- For each module:
    - Design concept. Contains technical solution concept and coding instructions
    - Review records. Contains a list of participants, findings, and resolutions
    - Test specifications for module test. Contains the concept of testing, automated and manual parts
    - Test protocols. Contain the test setup used and a verdict. If appropriate, execution logs are also included.

3.3 Product Quality

3.3.1 Automation in Development

In the effort to maintain a high overall product quality in an efficient manner, the key is to have a high degree of automation. Automation refers to the following steps:

- Immediate verification of coding results: This is typically implemented in the IDEs running automated unit test code and static analysers on the source code. If all tests at this level are passed, the software gets forwarded to the next stage.
- Continuous integration: All development results are transferred to a dedicated build and test environment. On a daily basis, the development results are compiled, and unit tests, integration tests and overall software tests are run automatically. Metrics and other analysis reports are gathered and provided as feedback to the developers. If all tests at this level are passed, the software gets forwarded to the next stage.
- Continuous deployment: All development results are transferred to a "QA stage", a computing environment as close to the final production environment as possible. The installation of the software in the QA stage is automated and triggered by successful integration builds. The software tests are run again, and additional (typically: manual) test steps are performed.
- Continuous delivery: If the quality assurance on the QA stage has run successfully, the software is automatically distributed to end-users.

The goal of this automation is to reduce the manual effort of testing as much as possible. Tests, refactorings and integration of additional functions will all be relatively painless, because the robust testing environment protects the existing functionality. openKONSEQUENZ will employ the first three steps, unit testing, continuous integration and continuous deployment.

3.3.2 Continuous Integration

Projects shall create test specifications which are run during continuous integration. The test cases shall determine, whether the integrated software's basic functionality is correct.
The continuous integration environment will calculate metrics, perform static and dynamic analysis, and will generate a feedback website showing the results of the tests and the metrics.

3.3.3 Continuous Deployment

After successful run of the continuous integration tests, the software build can automatically be forwarded to the QA environment. While continuous integration runs at least once per day, QA testing after deployment runs at least once per sprint.
Projects shall create test specifications for software/system test on the QA stage. This includes automated integration tests, (automated) UI-Tests, where appropriate and economically feasible, and descriptions for manual test steps.

3.3.4 Validation

Validation - the check whether or not the project delivered software like the PPC envisioned, and like the users require it - is performed on the QA environment. This means, that representatives from PPC or end users perform the validation. Projects shall create a validation specification that states, for each use case, the scenarios a end user should execute in order to validate the system behavior. Minimum required validation is a 3 months test operation on the QA environment. Additional validation measures have to be documented in the Validation concept.

Validation will also determine whether or not the project's outcome satisfies all regulations, e.g. the GUI-styleguide (see also the Architecture Committee Handbook, section 8.4, for usage of the styleguide document).

3.3.5 Document List

- Project test specification. Contains the strategy for testing the complete project outcome. This document also contains the description of manual tests. If the mock-up specifications (see 3., sect. “Verification”) do not cover all aspects needed for validation purposes, they have to be supplemented here.
- Validation concept. This documents contains a description of all manual tests to be executed by an end user in order to validate project outcome.
- The project shall deliver a “User documentation” (according to App. A, [3] “BDEW Whitepaper”, sect. 2.1.2.2)
- The project shall deliver a “Administration documentation” (according to App. A, [3] “BDEW Whitepaper”, sect. 2.1.2.2)

4. Development Setup

4.1 Environments

4.1.1 Development Environment

The environment for development is not regulated. It is recommended, that development environments use syntax checkers and static analyzers (e.g. PMD, Checkstykle, JSLint, …), but it is neither prescribed nor will it be checked. Development Environments shall use and run the Unit Test frameworks. Development Environments shall use git and Maven.

4.1.2 Integration Environment

The integration environment uses git and Maven to access the code and to build/test/deploy the software. It runs gerrit as a tool for reviews, Hudson as a continuous integration platform, SonarQube for metrics, and Nexus for software distribution. The integration environment also runs the Bugtracker (Bugzilla).

4.1.3 Quality Assurance Environment

The Quality Assurance (QA) environment uses the Nexus of the Integration Environment to get access the software. It uses Maven to run the automated software tests. Other than this, the QA Environment is similar to the production environments.

4.2 Development Tools

This section lists a set of development tools approved for use. The actual use of tools in a project is highly dependent on the technology of the project. Tools can be added in agreement with the AC.

4.2.1 Build automation

- Maven: https://maven.apache.org/

4.2.2 Source code management

- Git: https://git-scm.com/

4.2.3 Source code conventions and static analysis

Used for source code checks on development and integration environment.

- PMD: https://pmd.github.io/
- Checkstyle: http://checkstyle.sourceforge.net/
- Findbugs: http://findbugs.sourceforge.net/
- JSLint: http://www.jslint.com/

4.2.4 Automated code review tracking

- Gerrit: https://www.gerritcodereview.com/

4.2.5 Issue tracking

- Bugzilla: https://www.bugzilla.org/

4.2.6 Continuous integration

- Hudson: http://hudson-ci.org/

4.2.7 Source code and test coverage analysis

- SonarQube: http://www.sonarqube.org/

4.2.8 Software distribution and dependency management

- Nexus: http://www.sonatype.com/download-oss-sonatype

Appendix

A. Statement of Quality

The oK quality criteria are described in the following table:

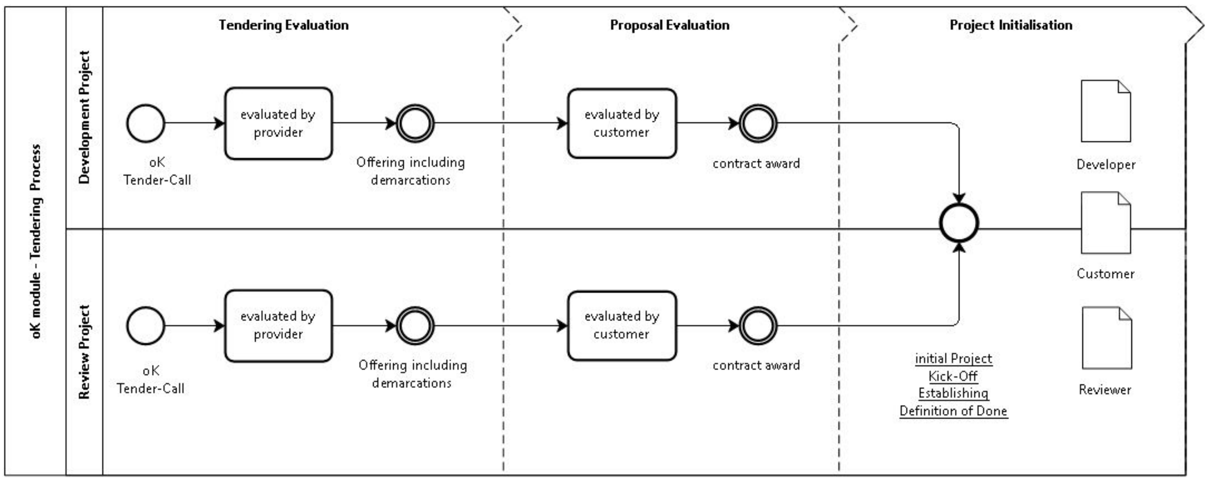
Quality criterion	**Subitem**	**Rating** [1-5] 1 - less important 5 - very important	**Scenarios**
Functional capability	Functional completeness		
	Functional accuracy		
	Functional adequacy		
Performance	Time response		
	Resource consumption		
	Capacity		
Usability	Appropriateness recognisability		
	Learnability		
	Controllability		
	Fail safety		
	Aesthetics of user interface		
	Accessibility		
Reliability	Maturity		
	Availability		
	Fault tolerance		
	Recoverability		

Table: Page 2

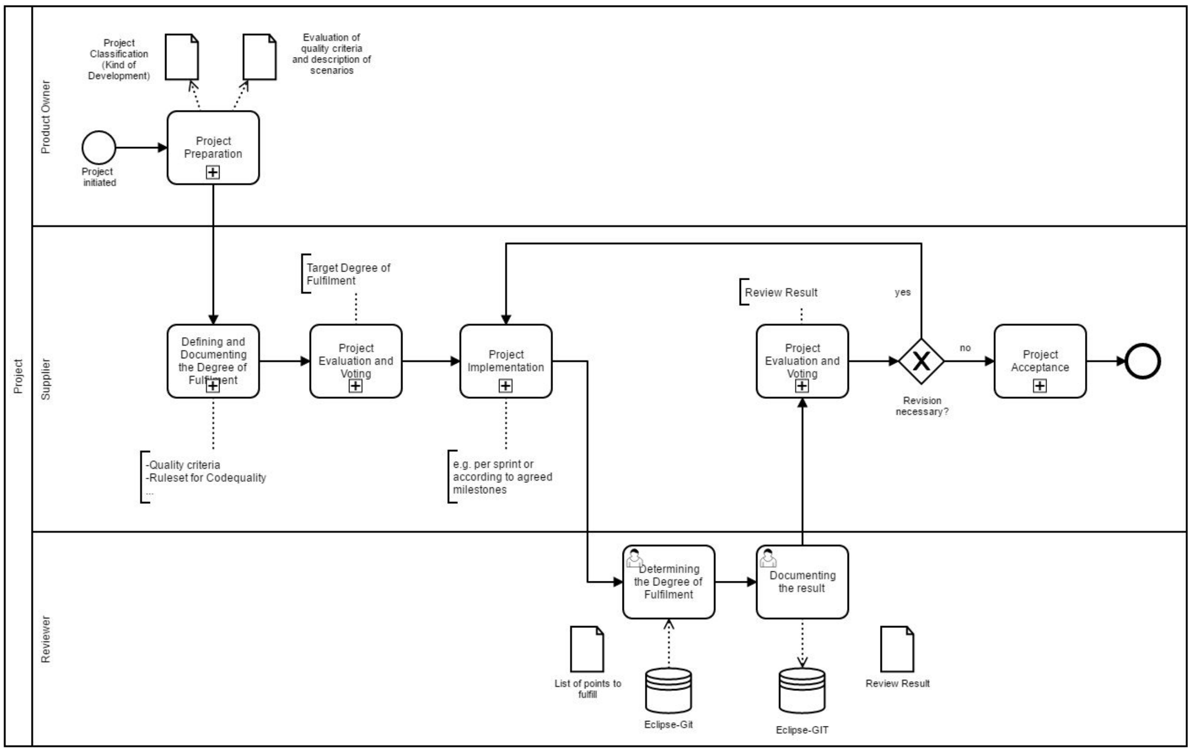
Quality criterion	**Subitem**	**Rating** [1-5] 1 - less important 5 - very important	**Scenarios**
Security	Confidentiality		
	Integrity		
	Verifiability		
	Responsibility		
	Authenticity		
Serviceability	Modularity		
	Reusability		
	Analysability		
	Modifiability		
	Testability		
Transferability	Adaptability		
	Installability		
	Interchangeability		

B. Processes

Project Initialisation

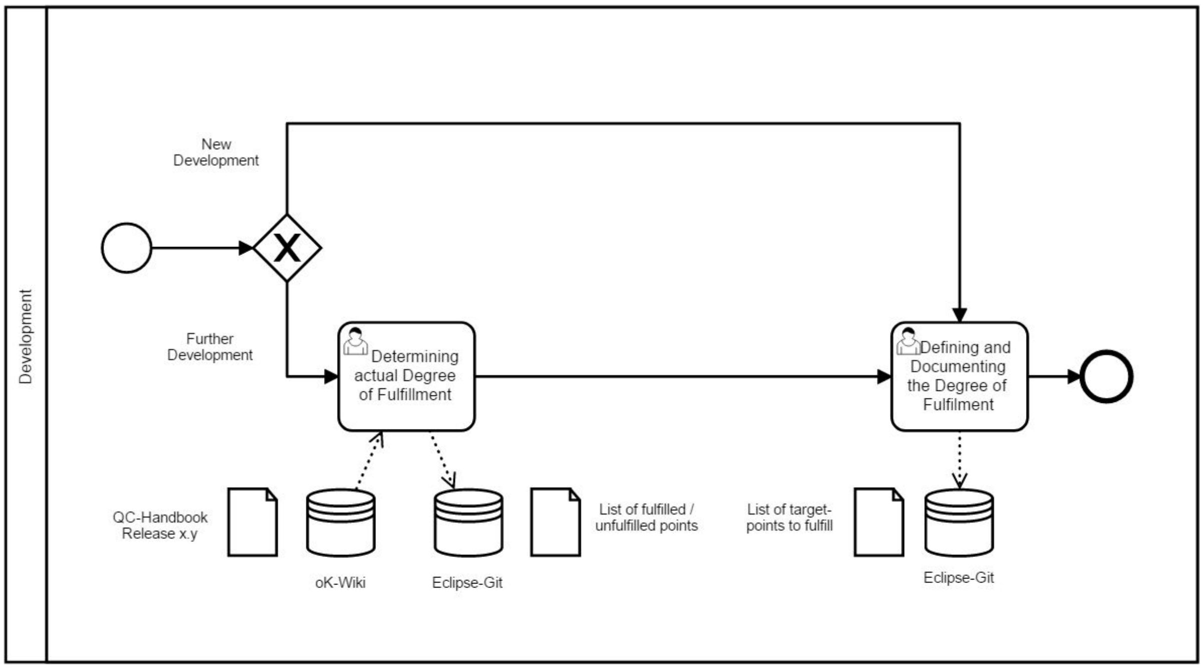


Project Procedure

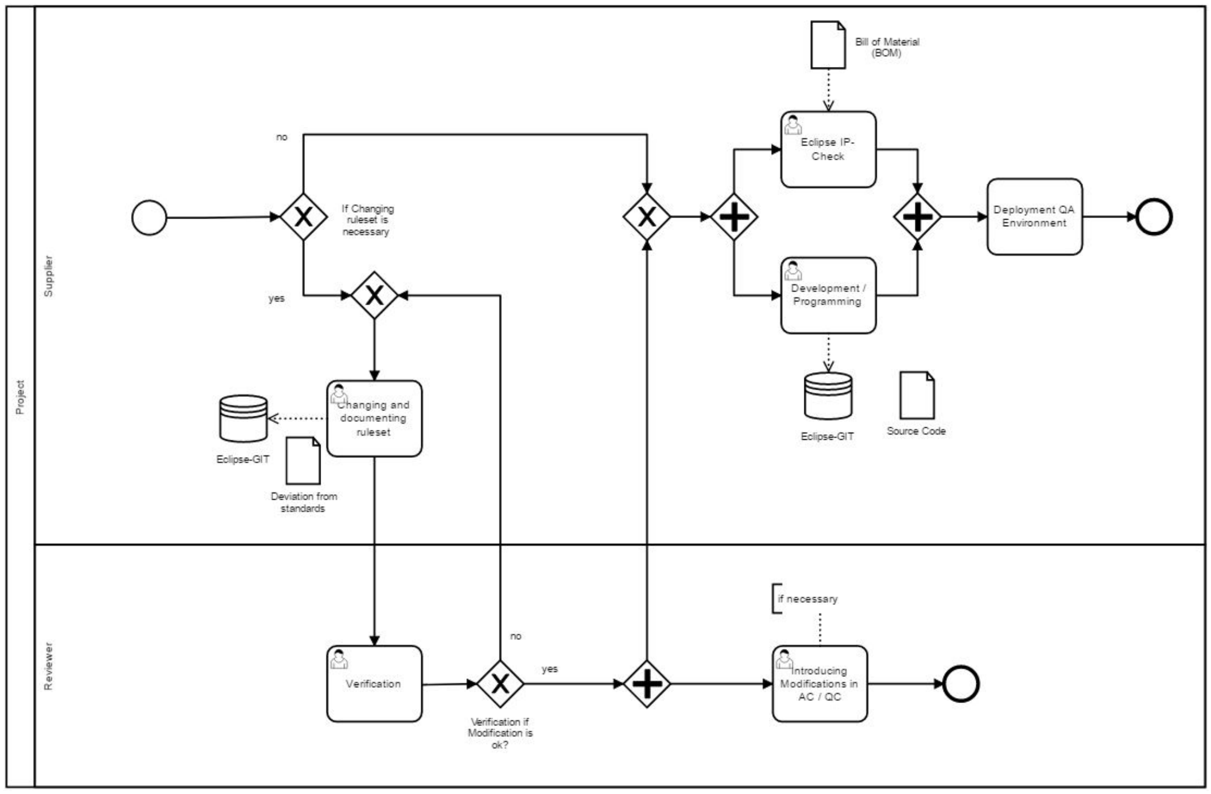


Project
Product Owner
Supplier
Reviewer
Project Classification (Kind of Development)
Evaluation of quality criteria and description of scenarios
Project initiated
Project Preparation
Target Degree of Fulfilment
Review Result
yes
no
Defining and Documenting the Degree of Fulfilment
Project Evaluation and Voting
Project Implementation
Project Evaluation and Voting
Revision necessary?
Project Acceptance
-Quality criteria
-Ruleset for Codequality
...
e.g. per sprint or according to agreed milestones
Determining the Degree of Fulfilment
Documenting the result
List of points to fulfill
Eclipse-Git
Eclipse-GIT
Review Result

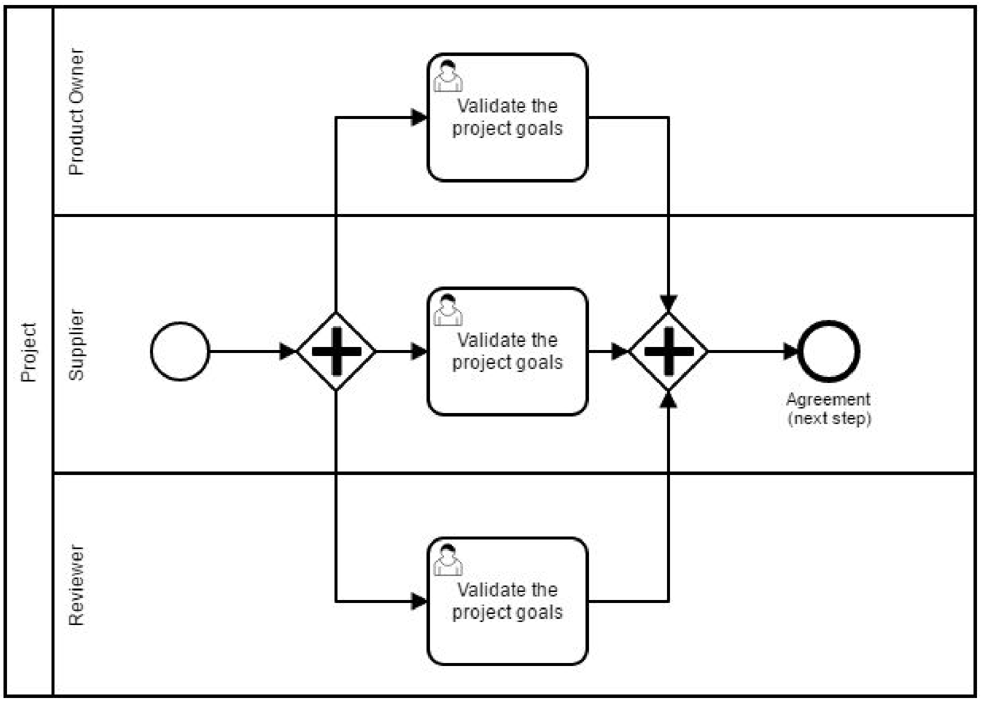
Defining and Documenting the Degree of Fulfilment


Development
New Development
Further Development
Determining actual Degree of Fulfillment
Defining and Documenting the Degree of Fulfilment
QC-Handbook Release x.y
oK-Wiki
Eclipse-Git
List of fulfilled / unfulfilled points
List of target-points to fulfill
Eclipse-Git

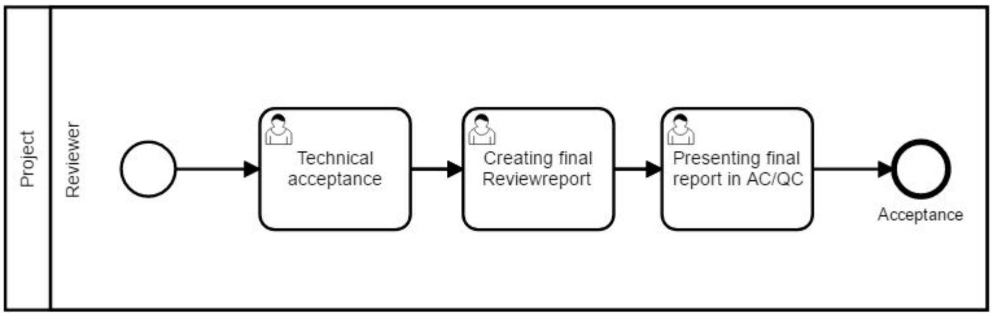
Project Implementation


Project
Supplier
Reviewer
no
If Changing ruleset is necessary
yes
Changing and documenting ruleset
Eclipse-GIT
Deviation from standards
Bill of Material (BOM)
Eclipse IP-Check
Development / Programming
Eclipse-GIT
Source Code
Deployment QA Environment
Verification
no
yes
Verification if Modification is ok?
if necessary
Introducing Modifications in AC / QC

Project Validation


Project
Product Owner
Supplier
Reviewer
Validate the project goals
Validate the project goals
Validate the project goals
Agreement (next step)

Project Acceptance



The acceptance stage is facilitated by the following review criteria based on existing empirical data:

Example

Checklist for Acceptance Procedures “Definition of Done”:

Checklist item	**Result**
The code has been completed	
Legal header added	
Unit tests produced / updated	
Code checked in in the eclipse-repository	
Quality gate (Sonarqube) passed	
Integration tests passed	
Regression tests passed	
Manual tests passed	
Module is installed in the QA environment	
IP check set up in Eclipse	
Interface documentation updated	
Architecture documentation updated	
Stipulated acceptance criteria met	

C. Project directory layout

The following directory structure and file name conventions have to be used (*adoc are asciidoc files). The files mentioned explicitly contain the documentation described in Chapter II “Quality Rules and Guidelines”. Other files (e.g. code and parameter files) have to be stored according to the maven standard directory layout.

```
/configuration/* (config files, esp. A configuration file for QA-environment)
/dependencies/bom.xml
/src/main/doc/get_started.adoc (how to get started with the project, where to
look, what to do)
/src/main/doc/arch/architecture.adoc (architecture concept and mapping to overall
architecture)
/src/main/doc/arch/model/* (UML tool files)
/src/main/doc/arch/images/*.png (UML diagram exports and other image files)
/src/main/doc/documentation/user.adoc (user documentation according to 4. Product
Quality)
/src/main/doc/documentation/admin.adoc (admin documentation according to 4.
Product Quality)
/src/main/doc/howto/build.adoc (how to build the software)
/src/main/doc/howto/code.adoc (how to set up for coding)
/src/main/doc/howto/config.adoc (how to set config parameters, semantics of
parameters)
/src/main/doc/howto/run.adoc (how to run a demo or the application)
/src/main/doc/howto/test.adoc (how to set up and execute tests)
/src/main/doc/variations.adoc (Abweichungen zum oK-Standard)
/src/main/reviews/YYYYMMDD-scope-type-author.adoc (review records)
                    (e.g. 20160429-ControllerComponent-Peer-ScroogeMcDuck.adoc)
/src/test/doc/test/integrationtest.adoc (integration test concept for the project)
/src/test/doc/test/test.adoc (software test concept for the project)
/src/test/doc/test/validation.adoc (validation concept for the project)
/src/test/protocols/YYYYMMDD-type-tester.adoc  (test execution records)
                    (e.g. 20160501-Integration-DonaldDuck.adoc)
/dirX (directory for module X)
/dirX/* (follow maven conventions from here)
/dirX/src/main/doc/arch/design.adoc (design and mapping to project architecture)
/dirX/src/main/doc/arch/model/* (UML tool files)
/dirX/src/main/doc/arch/images/* (UML diagram exports and other image files)
/dirX/src/main/doc/reviews/YYYYMMDD-scope-type-author.adoc (review records)
                    (e.g. 20160429-ControllerComponent-Peer-ScroogeMcDuck.adoc)
/dirX/src/test/doc/specs/moduletest.adoc (test concept for the module)
/dirX/src/test/<technology>/* (test code, e.g. JUnit code in
/dirX/src/test/java/*)
/dirX/src/test/protocols/YYYYMMDD-scope-type-tester.adoc (test exec records)
                    (e.g. 20160430-ControllerComponent-Module-DonaldDuck.adoc)
```

The names of the modules (here: “dirX”) have to be taken from the overall architecture definitions.

Example

The following describes a specific structure for a Java project based on the module org-eclipse-openk-domain-dynamictopology. The structure is described under the directory:

```
org-eclipse-openk-domain-dynamictopology

/pom.xml
/configuration/qa/dynamic-topology-service.properties
/configuration/qa/dynamic-topology-service.config
/dependencies/bom.xml
/src/main/doc/get_started.adoc
/src/main/doc/arch/architecture.adoc
/src/main/doc/arch/model/*
/src/main/doc/arch/images/*.png
/src/main/doc/documentation/user.adoc
/src/main/doc/documentation/admin.adoc
/src/main/doc/documentation/images/*.png
/src/main/doc/howto/build.adoc
/src/main/doc/howto/code.adoc
/src/main/doc/howto/config.adoc
/src/main/doc/howto/run.adoc
/src/main/doc/howto/test.adoc
/src/main/doc/howto/images/*.png
/src/main/doc/variations.adoc (Abweichungen zum oK-Standard)
/src/main/reviews/20180101-Model-peer-DonaldDuck.adoc

/src/test/doc/test/integrationtest.adoc
/src/test/doc/test/test.adoc
/src/test/doc/test/validation.adoc
/src/test/protocols/20180101-Integration-DonaldDuck.adoc

/service/pom.xml
/service/src/main/doc/arch/design.adoc
/service/src/main/doc/arch/model/*
/service/src/main/doc/arch/images/*.png
/service/src/main/java/org/eclipse/openk/domain/dynamictopology/service/*.java

/service/src/test/doc/specs/moduletest.adoc
/service/src/test/java/org/eclipse/openk/domain/dynamictopology/service/*Test.java
/service/src/test/protocols/20180101-Integration-DonaldDuck.adoc
```

D. Coding Guidelines

The overall technology stack of openKonsequenz is not yet defined. The following guidelines are subject to change. (In the future, they may be put in another document for easier version control.) The following list of coding guidelines must be adhered to:

Java

- https://google.github.io/styleguide/javaguide.html
- www.securecoding.cert.org/confluence/display/java
- http://www.onjava.com/pub/a/onjava/2003/11/19/exceptions.html
- https://www.javacodegeeks.com/2011/01/10-tips-proper-application-logging.html

JavaScript

- https://google.github.io/styleguide/javascriptguide.xml
- http://www.w3schools.com/js/js_conventions.asp

SQL

- http://www.sqlstyle.guide/
- https://google.github

XML

- .io/styleguide/xmlstyle.html

JSON

- https://google.github.io/styleguide/jsoncstyleguide.xml

The following list of metrics have to be calculated during the development:

- Size of code base
- Comment ratio
- Cyclomatic complexity
- Test coverage (line and branch)

Additional pointers to good practice:

- https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project#tab=OWASP_Top_10_for_2013
- https://cwe.mitre.org/top25/index.htm

Naming Convention for artifacts

To comply with the eclipse guidelines and to avoid name collisions the following rules should be considered. They can be used to determine the designators for:

- Java projects
- Java packages
- Java main-classes
- maven-artifacts (groupId, artifactId, name)
- paths (file-system)

For The placeholder “com.company” can be replaced by your company prefix.

Common Java artifacts

	rule	examples
prefix	`{{[com.company | org.eclipse.openk]}}`	• `com.company` • `org.eclipse.openk`
group-id	`{{prefix}}`	• `com.company` • `org.eclipse.openk`
artefact-id	`common`	• `com.company.common` • `org.eclipse.openk.common`
name	`{{group-id}}.{{artefact-id}}`	• `com.company.common` • `org.eclipse.openk.common`
package (without '-')	`{{prefix}}.{{artefact-id}}`	• `com.company.common` • `org.eclipse.openk.common`

CIM Versions

	rule	example
prefix	`{{[com.company|org.eclipse.openk]}}.cim`	• `com.company.cim` • `org.eclipse.openk.cim`
group-id	`{{prefix}}`	• `com.company.cim` • `org.eclipse.openk.cim`
artefact-id	`cim{{major-version}}v{{minor-version}}`	• `cim16v26a` • `cim17v07`
name	`{{group-id}}.{{artefact-id}}`	• `com.company.cim.cim16v26a` • `org.eclipse.openk.cim.cim17v07`
package (without '-')	`{{prefix}}.{{artefact-id}}`	• `com.company.cim.cim16v26` • `org.eclipse.openk.cim.cim17v07`

CIM Profiles

	rule	example
prefix	`{{[com.company | org.eclipse.openk]}}.cim.profile`	• `com.company.cim.profile` • `org.eclipse.openk.cim.profile`
group-id	`{{prefix}}`	• `com.company.cim.profile` • `org.eclipse.openk.cim.profile`
artefact-id	`{{profile}}`	• `entso-e` • `openkonsequenz`
name	`{{group-id}}.{{artefact-id}}`	• `com.company.cim.profile.entso-e` • `org.eclipse.openk.cim.profile.openkonsequenz`
path	`{{name}}`	• `com.company.cim.profile.entso-e` • `org.eclipse.openk.cim.profile.openkonsequenz`
package (without '-')	`{{prefix}}.{{artefact-id}}`	• `com.company.cim.profile.entsoe` • `org.eclipse.openk.cim.profile.openkonsequenz`

oK Service Artifacts

	rule	example
prefix	`{{[com.company | org.eclipse]}}.openk`	• `com.company.openk` • `org.eclipse.openk`
module group	`({{owner}} != "oK") ? {{owner}}-service : service`	• `company-service` • `service`
component level	`[common | core | model | logic | adapter | infrastructure | service]`	• `adapter`
group-id	`{{prefix}}.{{module group}}`	• `com.company.openk.company-service` • `org.eclipse.openk.service`
artefact-id	`{{module group}}-{{component level}}`	• `company-service-model` • `service-model`
name	`{{group-id}}.{{artefact-id}}`	• `com.company.openk.company-service.company-service-model` • `org.eclipse.openk.service.service-model`
path	`{{group-id}}/{{component level}}`	• `com.company.openk.btc-service/model` • `org.eclipse.openk.service/model`
package (without '-')	`{{group-id}}.{{component level}}`	• `com.company.openk.companyservice.model` • `org.eclipse.openk.service.model`
app-main-class (CamelCase)	`{{owner}}Service`	• `CompanyService` • `Service`

oK Core Modules

	rule	example
prefix	`{{[com.company | org.eclipse]}}.openk.core`	• `com.company.openk.core` • `org.eclipse.openk.core`
module group	`({{owner}} != "oK") ? {{owner}}-{{module-name}} : {{core-module}}`	• `company-module-name` • `module-name`
component level	`[common | core | model | logic | adapter | infrastructure | service]`	• `adapter`
group-id	`{{prefix}}.{{module group}}`	• `com.company.openk.core.company.module-name` • `org.eclipse.openk.core.module-name`
artefact-id	`{{module group}}-{{component level}}`	• `company-module-name-service` • `module-name-service`
name	`{{group-id}}.{{artefact-id}}`	• `com.company.openk.core.company-module-name.company-module-name-service` • `org.eclipse.openk.core.module-name.module-name-service`
path	`{{group-id}}/{{component level}}`	• `com.company.openk.core.company-module-name/service`
package (without '-')	`{{group-id}}.{{component level}}`	• `com.company.openk.core.modulename.service` • `org.eclipse.openk.core.modulename.service`
app-main-class (CamelCase)	`{{module group}}Service`	• `CompanyModuleNameService` • `ModuleNameService`

oK User Modules

	rule	example
prefix	`{{[com.company | org.eclipse]}}.openk.app`	• `com.company.openk.app` • `org.eclipse.openk.app`
module group	`({{owner}} != "oK") ? {{owner}}-{{app-name}} : {{app-name}}`	• `company-app-name` • `app-name`
component level	`[common | core | model | logic | adapter | infrastructure | service]`	• `adapter`
group-id	`{{prefix}}.{{module group}}`	• `com.company.openk.app.company-app-name` • `org.eclipse.openk.app.app-name`
artefact-id	`{{module group}}-{{component level}}`	• `company-app-name-adapter` • `app-name-adapter`
name	`{{group-id}}.{{artefact-id}}`	• `com.company.openk.app.company-app-name.company-app-name` • `org.eclipse.openk.app.app-name`
path	`{{group-id}}/{{component level}}`	• `com.company.openk.app.company-app-name/adapter` • `org.eclipse.openk.app.app-name/adapter`
package (without '-')	`{{group-id}}.{{component level}}`	• `com.company.openk.app.companyappname.adapter` • `org.eclipse.openk.app.appname.adapter`
app-main-class (CamelCase)	`{{module group}}Service`	• `CompanyAppNameService` • `AppNameService`

oK Source System

	rule	example
prefix	`{{[com.company | org.eclipse]}}.openk.source-system`	• `com.company.openk.source-system` • `org.eclipse.openk.source-system`
module group	`({{customer-name}} != "oK") ? {{customer-name}}-{{source-system-name}} : {{source-system-name}}`	• `customer-name-topology` • `mock-up-topology`
component level	`[common | core | model | logic | adapter | infrastructure | service]`	• `adapter`
group-id	`{{prefix}}.{{module group}}`	• `com.company.openk.source-system.customer-name-topology` • `org.eclispe.openk.source-system.mock-up-topology`
artefact-id	`{{module group}}-{{component level}}`	• `customer-name-topology-adapter` • `mock-up-topology-adapter`
name	`{{group-id}}.{{artefact-id}}`	• `com.company.openk.source-system.customer-name-topology.customer-name-topology-adapter` • `org.eclispe.openk.source-system.mock-up-topoloy-adapter`
path	`{{group-id}}/{{artefact-id}}`	• `com.company.openk.source-system.customer-name-topology/ adapter` • `org.eclipse.openk.source-system.mock-up-topology/ adapter`
package (without '-')	`{{group-id}}.{{component level}}`	• `com.company.openk.sourcesystem.customernametopology.adapter` • `org.eclispe.openk.sourcesystem.mockuptopology.adapter`
app-main-class(CamelCase)	`{{module group}}Service`	• `CustomerNameTopologyService` • `MockUpTopologyService`