DLR Software Engineering Guidelines

Version: 1.0.0

DLR

Revision Status

Version	Date	Remark
1.0.0	17.08.2018	Initial public version.

Table of Contents

1 Introduction

This document describes the software engineering guidelines of the German Aerospace Center (DLR). The target group of the guidelines are DLR scientists. The guidelines shall support them to find out the status of their developed software and to improve it with regard to good software development and documentation practice. The focus of the guidelines is on retaining knowledge and supporting sustainable software development in research.
The guidelines have been developed in cooperation with the members of the DLR software engineering network. The network is DLR's central exchange forum concerning software engineering. We publish these guidelines to support the general discussion about good software development practice in research.

1.1 Acknowledgement

The authors thank all persons involved and, particularly, the members the DLR software engineering network for their contributions. In addition, we would like to thank DLR's central IT department for their ongoing financial support for this important topic.

1.2 Further Information

In the following, you can find further information about the guidelines and the overall concept:

- T. Schlauch, C. Haupt, "Helping a friend out. Guidelines for better software", Second Conference of Research Software Engineers, September 2017. [Online]. Available: https://elib.dlr.de/114049/
- T. Schlauch, "Software engineering initiative of DLR: Supporting small development teams in science and engineering", ESA S/W Product Assurance and Engineering Workshop 2017, September 2017. [Online]. Available: https://elib.dlr.de/117717/
- C. Haupt, T. Schlauch, "The software engineering community at DLR: How we got where we are" in Workshop on Sustainable Software for Science: Practice and Experiences (WSSSPE5.1), N. C. Hong, S. Druskat, R. Haines, C. Jay, D. S. Katz, and S. Sufi, Eds., September 2017. [Online]. Available: https://elib.dlr.de/114050/
- C. Haupt, T. Schlauch, M. Meinel, "The software engineering initiative of DLR - overcome the obstacles and develop sustainable software" in 2018 ACM/IEEE International Workshop on Software Engineering for Science, June 2018. [Online]. Available: https://elib.dlr.de/120462/

1.3 Citation

T. Schlauch, M. Meinel, C. Haupt, "DLR Software Engineering Guidelines", Version 1.0.0, August 2018. [Online]. Available: https://doi.org/10.5281/zenodo.1344612

1.4 Licence

1.5 German Language Edition

The original German document is available via: https://doi.org/10.5281/zenodo.1344608

2 Terms and Abbreviations

2.1 Terms

Software	The term software generally refers to programs that are run on a computer or similar devices. In addition to the program, the software includes, for example, the source code, user documentation, test data and the architectural model.
Software with product characteristics	Software that must be used and operated in a productive context. Possibly, this software is an essential part of a cooperation with other organisations.
Software Responsible	The software responsible has the technical and professional overview of a software. For software with limited scope, this is usually the current main developer.
persons involved in the development	In this document the phrase "persons involved in the development" refers to all persons directly contributing to the development of the software. These are, for example, the software developers or testers. Alternatively, the term "development team" is used as well.
SoftwareEngineering.Wiki	The SoftwareEngineering.Wiki is the central DLR-internal Wiki space to exchange software engineering related information and knowledge.

2.2 Abbreviations

AC	Application class
EQA	Recommendation "Qualification"
EAM	Recommendation "Requirements Management"
ESA	Recommendation "Software Architecture"
EÄM	Recommendation "Change Management"
EDI	Recommendation "Design and Implementation"
EST	Recommendation "Software Test"
ERM	Recommendation "Release Management"
EAA	Recommendation "Automation and Dependency Management"

3 Application Classes

The application classes (AC) help to define appropriate software quality measures. They allow activities and tooling to be adapted to needs and to structure stakeholder communication with regard to software quality.
The application classes define recommendations based on each other to ensure appropriate engineering practice and software quality. They primarily address investment protection, risk reduction and knowledge retention. The measures taken must be geared to the requirements of the application class.
The application classes primarily support the development of individual software in the facilities. In addition, they can be used as a basis for requirements towards externally commissioned companies to ensure the quality of the development. This is particularly recommended if the externally created software is to be maintained or further developed by the facility at a later stage.

3.1 Application Class 0

For software in this class, the focus is on personal use in conjunction with a small scope. The distribution of the software within and outside DLR is not planned.
Software corresponding to this application class frequently arises in connection with detailed research problems. The respective facility specifies the necessary measures for this application class itself, for example, for observance of good scientific practice. Examples of a possible classification in application class 0 are:

- Scripts to process data for a publication.
- Simple administrative scripts to automate specific tasks.
- Software that only demonstrates certain functions or is developed to test them.

3.2 Application Class 1

For software of this class, it should be possible, for those not involved in the development, to use it to the extent specified and to continue its development. This is the basic level to be strived for if the software is to be further developed and used beyond personal purposes.
For this purpose, the current version must be traceable and reproducible. It is necessary that the basic requirements and constraints, the available functional scope as well as known problems of the software are evident.
This application class is recommended if the software does not offer a wide range of functions or if the facility only develops it within a narrow scope. Examples of a possible classification in application class 1 are:

- Software that students develop during studies, bachelor or master theses.
- Software resulting from dissertations in which the long-term development does not matter.
- Software resulting from third-party projects with focus on demonstration and without planned long-term development.

3.3 Application Class 2

For software in this class, it is intended to ensure long-term development and maintainability. It is the basis for a transition to product status.
This requires the structured management of the respective requirements. In particular, the constraints and quality requirements must be addressed by an appropriate software architecture. This describes the technical concepts and the structure of the software, ensures that development know-how is preserved, and makes it possible to assess the suitability for new usage scenarios. Furthermore, a defined development process, rules for design and implementation as well as the use of test automation are essential in this context.
This application class is recommended if the software offers a wide range of functions and the facility develops it long-term. Examples of a possible classification in application class 2 are:

- Software resulting from dissertations in which maintainability and long-term usage matter.
- Software from third-party projects in which maintenance and long-term usage matter beyond the project.
- Large research frameworks which are developed by a majority of a department (without product characteristics).

3.4 Application Class 3

For software in this class, it is essential to avoid errors and to reduce risks. This applies in particular to critical software and that with product characteristics.
To this end, active risk management has to be carried out. I.e., risks of the technical solution must be actively identified and addressed in the software architecture. In addition, by expanding test automation and structured reviews, errors should be detected at an early stage to ideally prevent them in a production version. Furthermore, traceability of changes has to be ensured.
This application class is recommended if the development entails high risks for the facility. These can arise, for example, from product liability, certification, external requirements or the importance of the software for value-adding activities. Examples of a possible classification in application class 3 are:

- Mission critical software, for example, in context of aircraft, autonomous vehicles or space missions.
- Software for which the facility gives a warranty within or outside DLR (e.g., via an external company).
- Software that makes a significant contribution to third party funding and research results of the facility, and must therefore work reliably.

3.5 Definition of the intended Application Class

At the start of development, the software responsible, along with other specialist parties involved if necessary, determines the intended application class. In addition, it is regularly checked whether the classification of the intended application class should be changed.

The decision criteria for assigning an application class are directly aligned with the objectives of the respective application class. The criteria result in a decision tree (see Figure 1). This is merely a recommendation, which allows for justified deviation.
The classification in an application class is explained below on the basis of the criteria:

1. **Risks for the facility:** This is the first and foremost important decision-making criterion. High risks for the facility can arise, for example, from product liability, certification, external requirements or the importance of the software for the value-adding activities. A "failure" of the software could thus result in sensitive cutbacks for the facility or a part of it. Therefore, in cases of high risk, it is recommended to strive for classification in application class 3, regardless of the other criteria.
2. **Scope:** The next criterion is the expected scope. For a small scope, either application class 0 or 1 is sufficient. This also applies in case of planned long-term usage and development of the software. An objective assessment of a small scope is difficult. Metrics such as "lines of code" can only be correlated to a certain extent with the scope. Limiting the total development effort is therefore recommended. For a small scope, the effort to implement the software (including the implementation of the recommendations of application class 1) should not exceed one person year.
3. **Distribution of the software:** This criterion refers to the distribution of the software within and outside the DLR. In particular, licensing aspects must generally be considered if the software is distributed to third parties outside DLR. In these cases, application class 1 at least must be selected. If the software is not used by other colleagues, classification in application class 0 is sufficient.
4. **Period of further development:** This criterion refers to the expected period during which the facility will develop and maintain the software. If there is a large functional scope and development has to be ensured over an extended period of time, application class 2 has to be applied. In this case, there is an increased need to counter the loss of know-how. A longer period of development applies, if development is to be continued after the possible leave of important know-how carriers (> 2 years).

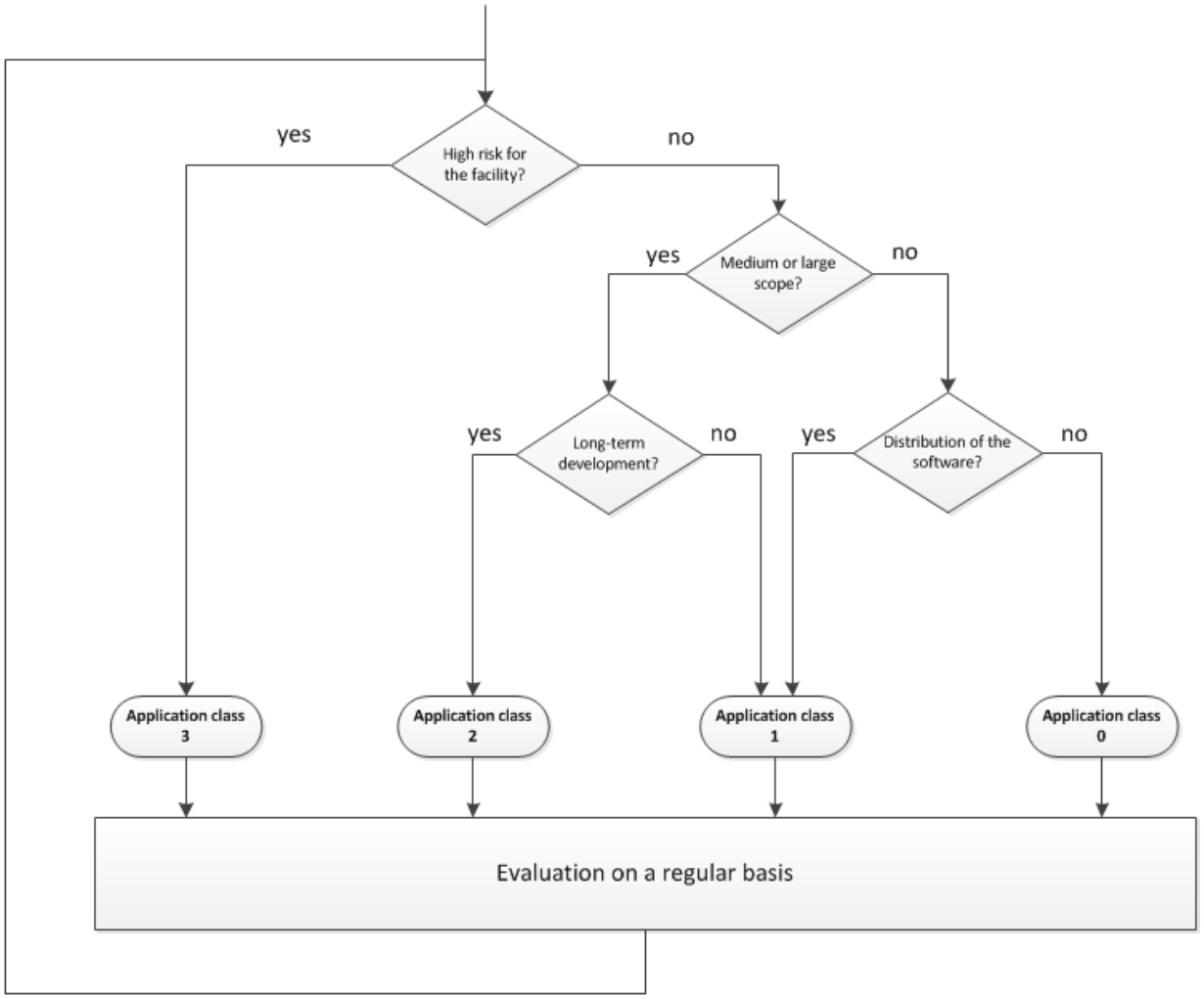

Figure 1: Decision tree for the determination of the intended application class

3.6 Evaluation of the achieved Application Class

On the basis of the intended application class, the software responsible assesses the application class achieved by the software. To this end, it has to be determined to which extent the recommendations of the respective application class have been implemented. For this purpose, in addition to this document, checklists for application classes 1 - 3 are provided in various formats. They list all the recommendations relevant for an application class. However, it may be useful to implement the recommendations of a higher application class, at least in a weaker form. This simplifies the transition to a higher application class. The next section provides a detailed overview of all recommendations including explanations and additional information.
The recommendations should be interpreted and evaluated within the context of the corresponding facility. If the desired application class is not achieved, the software responsible, if necessary in alignment with other specialist parties involved, determines appropriate measures for further development. In this context, the cost-benefit ratio must be considered realistically given the remaining development time and resources.

4 Overview of the Recommendations

This chapter describes the recommendations for various software development areas. At the beginning of every section, the content is summarised and significant terms are introduced. This is followed by recommendations including explanations. For every recommendation, the application class for which it applies is stated (see the Application Classes section). The sorting of the recommendations is based on the structure of the introductory section.
The implementation of the individual recommendations is kept deliberately open to allow the best possible decision in dependence of the respective development context. However, there are normally initial hints on this in the explanatory text. The *SoftwareEngineering.Wiki* provides further information, tool and literature recommendations, as well as concrete examples for the various software engineering topics.

4.1 Qualification

The following recommendations are intended to ensure that persons involved in the development have the necessary knowledge and training. Filling the existing gaps through training is recommended. This must be considered, among other things, during the preparation of the individual training plans.

Recommendation	from AC	Explanation
EQA.1: The software responsible recognises the different application classes and knows which is to be used for his/her software.	1	This knowledge is the prerequisite for implementing the measures recommended at DLR to ensure good engineering practice and software quality.
EQA.2: The software responsible knows how to request specific support at the beginning and during development as well as to exchange ideas with other colleagues on the subject of software development.	1	The knowledge of other contacts on the subject of software engineering is important to solve problems more easily at the start and during development.
EQA.3: The persons involved in the development determine the skills needed with regard to their role and the intended application class. They communicate these needs to the supervisor.	1	At the level of application class 1, in addition to the technical knowledge, at least the knowledge of the programming language and a version control system is necessary. Higher application classes may require additional skills at the team level. These must be developed or trained in a focused manner.

EQA.4: The persons involved in the development are given the tools needed for their tasks and are trained in their use.	1	In particular, they have to master the handling of the tools in the respective scope of use. Otherwise, unnecessary effort and rework due to misunderstandings are likely.

4.2 Requirements Management

The central entry point into requirements management is the **problem definition**. It describes the objectives and the purpose of the software in a concise and comprehensible form. It also summarises the essential requirements. It thus defines "the why" and "the what" and serves as a guide for decisions.

A **requirement** describes a property to be fulfilled in the software. There are different types of requirements:

- **Functional requirements** describe desired functions of the software.
- **Quality requirements** describe expected qualitative characteristics of the software (e.g., usability, security, efficiency, see ISO/IEC 25010).
- **Constraints** describe limitations that have to be respected during the development and design of the software.

Requirements give direction to software development. In particular, quality requirements characterise the resulting solution approach and often involve risks. Therefore, quality requirements should be early coordinated and analysed. The desired **product quality of the software** must always be explicitly designed and is individually defined for each software.

Recommendation	from AC	Explanation
EAM.1: The problem definition is coordinated with all parties involved and documented. It describes the objectives, the purpose of the software, the essential requirements and the desired application class in a concise, understandable way.	1	It is important that the problem definition is early coordinated between the parties involved to prevent misunderstandings and incorrect developments. The problem definition also provides important hints for later use and further development.
EAM.2: Functional requirements are documented at least including a unique identifier, a description, the priority, the origin and the contact person.	2	Requirements must be clearly identifiable to refer to them during development and to trace them back to software changes (see the Change Management section). In addition, prioritisation helps to determine the order of implementation. Finally, information about the contact person and the origin is essential in case of questions.
EAM.3: The constraints are documented.	1	The relevant constraints (e.g., mandatory programming languages and frameworks, the opera-

		tional environment, legal aspects) should be early coordinated to avoid misunderstandings and incorrect developments. In addition, they allow justified decisions, because they help to rule out options. In case of software with limited scope, it is recommended to document constraints as part of the problem definition (see recommendation EAM.1).
EAM.4: The quality requirements are documented and prioritised.	2	The product quality of each software has to be considered individually. The relevant quality requirements should be early coordinated and defined. In particular, prioritisation is necessary, because quality characteristics partially conflict with each other. A good understanding of the quality requirements is essential to avoid misunderstandings and incorrect developments. Forgotten and missing quality aspects often result in major changes of the software. In practice, it is useful to concretise quality requirements with the help of scenarios. Such a quality scenario describes a typical usage scenario of the software, with the focus on a quality characteristic (e.g., "The system displays the first search results after one second."). As a result, required quality characteristics are more comprehensible, discussable and verifiable.
EAM.5: User groups and their tasks are documented in the respective usage context.	2	This analysis is essential to build up understanding for the users of the software and to create an appropriate solution. Without this analysis, the software is unlikely to be accepted, or effort is spent on functionalities that are not used in their implemented form.
EAM.6: Active risk management is carried out. The risks resulting from the development are documented with the probability of occurrence and the expected effects.	3	Risks arise, for example, from unclear, incomplete requirements and associated late changes. In addition, lack of know-how regarding the use of a technology or the technology itself can pose risks. This can lead to delays and additional effort. It is therefore important to identify and track risks actively and to take appropriate countermeasures (e.g., creation of prototypes, increasing know-

		how). In particular, risks related to the software architecture (see the Software Architecture section) must be taken into consideration, since these can potentially cause considerable damage.
EAM.7: Guidelines for the formulation and documentation of requirements are defined and consistently applied.	3	These guidelines ensure that all essential information is documented consistently and contribute to error prevention.
EAM.8: A glossary exists which describes the essential terms and definitions.	2	The glossary defines a common vocabulary. It helps to avoid misunderstandings and errors based on different terms and definitions.
EAM.9: The list of requirements is regularly coordinated, updated, analysed and checked. The resulting changes are traceable.	2	Requirement-related activities are performed continuously - especially before the start of a new development stage. In this context, it is important that stakeholders coordinate to develop a common understanding of the next steps and to further refine the requirements. This avoids misunderstandings and incorrect development based on non-compliant requirements or parts of them. In addition, a consistent list of requirements is obtained and contradictions in the requirements can be identified and resolved.
EAM.10: For each requirement, applied changes to all elements of the software (e.g., source code, test cases) can be traced (traceability).	3	This measure provides an understanding of the requirement impact. For example, it is possible to check that the required implementation including test cases exists for a requirement. In addition, errors can be narrowed down and associated with a requirement easily.

4.3 Software Architecture

Software architecture conveys an idea of the core components of the software, on which the rest of the software builds upon. A change in the core components will be expensive and endangers software quality.
Here is an example: A distributed system shall be realised. The quality requirements prescribe that the components shall communicate encrypted (security). It must also be ensured that no messages are lost (reliability). Furthermore, there are constraints that limit the choice of licences for third-party software. In this context, the communication technology is such a core component. A wrong decision can lead to non-conformance with the requirements regarding security and reliability. In addition, a later change in technology may lead to a high effort in adaptation. The decision-making process can therefore become quite time-consuming. Different options must be re-

searched, knowledge about alternative solutions must be built up, and prototype implementations may have to be created. Constraints, such as limits in choosing licences, help to exclude certain options.
It can be seen that software architecture can be systematically derived from the requirements. In particular, quality requirements often lead to architectural questions. Constraints alleviate choosing the appropriate solutions. Therefore, the concrete amount of architecture and effort are dependent on the development context. They heavily depend on the quality requirements and the experience of the participants.
For substantially large software, which is maintained and further developed over a longer period of time, the creation of **architecture documentation** pays off. Typically, it provides information about the software structure, how its parts interact, the overall concepts, and the key decisions. The concrete aspects and the depth of detail depend on the relevant target groups. The architecture documentation contains essential, conceptual development knowledge, which either may or may not be readily distilled from the source code. This knowledge is essential for maintaining the software efficiently and purposefully in the long term.
The following recommendations shall ensure that significant, overall decisions regarding the software are described and worked out in a structured manner.

Recommendation	from AC	Explanation
ESA.1: The architecture documentation is comprehensible for the relevant target groups.	2	Software architecture is an important entry point into the software for various stakeholders. For example, it provides important information for developers regarding structure, interfaces and architectural concepts that have to be adhered to. The customer gets an overview on how the software integrates into the rest of the system landscape and how the implementation of central qualitative properties is ensured. It is therefore important to identify the relevant target groups for the specific case and to prepare the information for them in a comprehensible manner.
ESA.2: Essential architectural concepts and corresponding decisions are at least documented in a lean way.	1	These are concepts and decisions not easily deducible from source code (e.g., "What are the domain-specific components and how do they work together?", "How does the overall parallelisation concept work?", "Why is a particular library used to connect an external system?"). This knowledge is important for efficient development of the software, especially if the previous main developer is no longer available.

ESA.3: Testability of the software is appropriately addressed at software architecture level.	2	The software (or a part of it) has to be brought into a defined state to execute a test case. In addition, it must be possible to observe relevant effects. These testability properties must be addressed in the software architecture adequately. It is therefore necessary to align the test strategy with the software architecture (see the Software Test section). For example, realising test interfaces and testing infrastructure needs to be foreseen and design principles (see the Design and Implementation section) that encourage a testable software structure need to be prescribed.
ESA.4: The software architecture is coordinated with the relevant target groups. Changes are communicated actively and are comprehensible.	2	A common understanding of central concepts of the solution shall be supported, so that decisions can be understood and implemented by all parties involved. In addition, it shall prevent overlooking important aspects. By that, further development and refinement of the solution concepts is visible and comprehensible for all participants.
ESA.5: The overlap between architectural documentation and implementation is minimised.	2	The architecture documentation must be updated along with ongoing development. Therefore, it is recommended to not include implementation details (e.g., the internal structure of a component) in it. Especially duplication of information needs to be avoided, since it cannot be kept synchronised in the long run.
ESA.6: The architecture documentation consistently uses the terminology of the requirements.	2	First and foremost, it makes it easier to get into the solution concepts. In addition, misconceptions and errors stemming from different interpretation of the terminology (see the Requirements Management section, recommendation EAM.8) are avoided.
ESA.7: Architectural concepts and decisions can be traced to requirements.	2	In particular, the qualitative characteristics that have to be met strongly influence many aspects of the software architecture. In addition, constraints help to select the appropriate solution from the various options. Finally, the requirement-specific approach helps to comprehend and justify the software architecture.

ESA.8: Key architectural concepts are checked for their suitability using appropriate methods.	2	Unfavourable decisions at the architectural level often lead to considerable extra effort. For this reason, the key concepts have to be tested practically - for example with the help of a prototype.
ESA.9: The architecture documentation is updated regularly.	2	The architecture documentation must be consistent with the current implementation of the software. Outdated or partially incorrect architecture documentation reduces its usefulness greatly and is sometimes more critical than no architecture documentation at all.
ESA.10: A systematic review of the software is carried out regularly to find deviations from the software architecture.	3	This makes unfavourable decisions visible at the architecture and implementation level. The resulting potential improvements are to be assessed and implemented, prioritised through the change process. The use of code analysis tools is recommended.
ESA.11: A systematic review of the software architecture is carried out on a regular basis to find out whether it meets the specified requirements.	3	It ensures that the software architecture adequately addresses the specified requirements. Unfavourable architectural decisions can be identified. The resulting potential improvements are to be assessed and implemented, prioritised through the change process.

4.4 Change Management

Change management is about performing changes to software in a systematic and comprehensible way. Software changes may be induced by requirements, bugs or optimisations. Change management helps to keep track of the overall development status and to coordinate the different development tasks.

In this context, the **change process** describes how **change requests** (e.g., requirements, errors, optimisations) are in general processed by developers, and possibly resulting in a new software version. When looking into details, this process is different from one development context to the other. It is therefore important to agree on it in the development team and to improve it continuously. In practice, it must be ensured that the procedures can be performed in an efficient way. It is therefore advisable to use tools and automation in an appropriate way.

To centrally document change requests, web-based **ticket systems** (e.g., MantisBT, Jira) - also known as "bug tracker" or "issue tracker" - are often used. This allows to keep track of all tasks to be resolved. Ticket systems allow assigning change requests to specific software versions. On this basis, they provide planning overviews (**roadmap**) and detailed change histories (**change log**). Finally, ticket systems often allow adapting the issue tracking procedure to the individual

change process. The use of a ticket system is particularly worth for longer term development of large software and if distributed development teams need to cooperate.
Another important task of the change management is to preserve the results of the development work in a safe and comprehensible way. The results are, for example, the source code, the test procedures including required test data or the user manuals. These results are typically stored in a project **repository**.
The repository hosts ideally all artefacts which are required to build an executable version of the software and to test it. The directories and files within the repository are managed by a **version control system** (e.g., Git, Subversion). It makes sure that any change to the repository (**commit**) is logged with a description (**commit message**) and can be listed in the version history. On this basis, it provides decisive advantages for the development. For example, you are able to restore old or already removed versions. In case of bugs, you are able to easier isolate the cause with the help of the change history. Important interim states can be defined (**tag**) and are quickly detectable. Parallel changes on the same files are detected and developers are assisted in resolving conflicts. Finally, different developer groups are able to work independently using parallel development **branches**. The additional effort to learn and use a version control system quickly pays off in practice.
The following recommendations shall ensure a structured way of handling software changes and their traceability.

Recommendation	from AC	Explanation
EÄM.1: The change process is coordinated in the development team and documented.	2	The change process describes the basic practical development procedures and embeds essential test activities (see the Software Test section). The process foremost supports developers in enhancing the cooperation within the development team and to avoid errors. In this context, attention must be paid to practicality. I.e., the development procedures should be well supported by tools and the automation of routine tasks (see the Automation and Dependency Management section). It is recommended to review the change process in regular intervals.
EÄM.2: The most important information describing how to contribute to development are stored in a central location.	1	This information is essential for new developers, or if development is resumed after a longer pause. It includes the basic steps required to start development (e.g., "What is necessary to create the executable software?", see the Automation and Dependency Management section, recommendations EAA.1 and EAA.2).

		This information is often directly available in the repository. It is typically stored in a file named "README" or "CONTRIBUTING". As alternative, a web page may be created for the introduction.
EÄM.3: Change requests are centrally documented at least including a unique identifier, a short description and the contact details of the originator. They are stored long term and are searchable. In the case of bug reports, additional information about the reproducibility, the severity and the affected software version shall be documented.	2	This way all tasks related to the software are available in one place. In addition, this approach makes sure that sufficient information is available to work on them. On this basis, all tasks can be surveyed and prioritised in a sensible way. The unique identifier allows referring to the tasks from another context or tool. This is the basis to trace changes from their source to their impacts (traceability).
EÄM.4: A planning overview (roadmap) exists describing which software versions shall be achieved by when with which results.	2	The roadmap provides a clear view on a potentially large number of change requests. It makes clear, which tasks are in the focus of the current development phase. Bottlenecks and content overlaps are easier to find. In practice, the roadmap supports the discussion of development progress and is an efficient tool for release planning (see the Release Management section).
EÄM.5: Known bugs, important unresolved tasks and ideas are at least noted in bullet point form and stored centrally.	1	This information simplifies the further development or is also interesting for users of the software. In the simplest case, it is kept as part of the "README" file in the repository.
EÄM.6: A detailed change history (change log) exists providing information about the functionalities and bug fixes of a software version.	2	The change log provides a clear view on a potentially large number of change requests. It is clear which functionality or bug fix is provided by which concrete software version. This is helpful to isolate the cause of bugs. Finally, release notes can be created on the basis of the change log easily (see the Release Management section).
EÄM.7: A repository is set up in a version control system. The repository is adequately structured and ideally contains all artefacts for building a usable software version and for testing it.	1	The repository is the central entry point for development. All main artefacts are stored in a safe way and are available at a single location. Each change is comprehensible and can be traced back to the originator. In addition, the version control system ensures the consistency of all changes.

		The repository directory structure should be aligned with established conventions. References are usually the version control system, the build tool (see the Automation and Dependency Management section) or the community of the used programming language or framework. Two examples: 1. The main development branch is called "trunk" when using the version control system Subversion. For Git, it is called "master". 2. The build tool Maven (see the Automation and Dependency Management section) largely standardises the directory structure in the main development branch. For example, the Java source code should be stored in the directory "src/main/java". Tests are located in the "src/test" folder. In particular, it is recommended to keep the directory structure below the main development branch stable. Build tools for automatic building and testing of the software rely on this. To reproduce intermediate software versions, it is necessary that all required artefacts are contained in the repository. In addition to the source code, this usually includes test scripts and test data (see the Software Test section) as well as dependencies, configuration settings and scripts for building the software (see the Automation and Dependency Management section). However, there are limitations in practice. For example, if the artefacts are very large or strongly dependent on the operating system. In such cases, at least sufficient information needs to be stored to access these artefacts if required.
EÄM.8: Every change of the repository ideally serves a specific purpose, contains an understandable description and leaves the software in a consistent, working state.	1	Version control systems help to recover past versions, to narrow down errors and to trace changes. To use these functions efficiently, the following procedure is recommended: A change in the repository ideally serves exactly one purpose. For example, it fixes a bug or adds a

		new functionality. If a commit mixes a lot of different changes, it will be harder to integrate a contained bug fix into other development branches. In addition, it is harder to trace back bugs to their original cause. The commit message describes the purpose of the change and briefly summarises the most important details. Information which can be directly retrieved from the version control system should not be repeated in the commit message. For example, a version control system shows all content modifications of a commit in a clear way. It is recommended to start the commit message with a short, significant sentence. Details can be added, separated by an empty line. This approach increases the clarity when working with the version history. After committing, the software remains in a working state. This avoids that other developers are hindered in their work. It also makes it easier to integrate a modification in other development branches.
EÄM.9: If there are multiple common development branches, their purpose can be identified easily.	2	This increases clarity in the repository. The main development branch (e.g., named "trunk" or "master") typically contains the latest version of the software. Other active development branches may exist, for example, to implement a certain functionality or to stabilise a software version prior release. In this context, it is recommended that only those development branches are visible on which work is really in progress. The format and the meaning of the development branch names should be defined. For example, the branch named "RB-1.0.0" serves to stabilise the production version 1.0.0 (see the Release Management section).
EÄM.10: For each change request, the modifications can be traced in the repository (traceability).	3	This allows, for example, to check whether all required modifications have actually been done. In addition, impacts of a change request are directly visible. In combination with the change log,

		searching the causes of bugs can become easier. In practice, this can be achieved by integrating the ticket system with the version control system. Each commit message typically contains a reference to a change request. This procedure allows to directly inspect, within the ticket system, the impact of change requests in the repository. In addition, the version history of the repository allows one to determine the cause of a modification.

4.5 Design and Implementation

An initial idea of the high-level software structure often emerges in the beginning of the development. It describes a suitable decomposition of the software in accordance to functional and technical aspects. In the following, as an example, we assume a decomposition of the software into **components**. On this basis, step by step, the parts of the components - here the **modules** - are designed and implemented. It is often desired that the software structure is comprehensible, easily changeable and extensible. In addition, it has to comply with the conceptual constraints of the selected software architecture (see the Software Architecture section). For example, a keyword search throughout a text has to be implemented. The selection of the search algorithm and other design decisions strongly depend on the required response time and the size of the text. Thus, knowledge about these constraints is essential to implement an appropriate search module. Design and implementation are aligned closely and performed in small, repeating steps. Continuous refactoring and testing at module level support this iterative approach. **Refactoring** is an improvement of the software structure while keeping its visible behaviour. This practice is essential to maintain the quality of the software structure in the long term. **Module tests** (see the Software Test section) form the basis for efficient refactoring. They allow to verify software changes quickly.

Design principles and patterns provide concrete suggestions for an appropriate implementation. **Design principles** are heuristics related to design and implementation. Their consistent application has shown positive effects on the quality of the software structure. For example, the Don`t-Repeat-Yourself principle recommends to ideally avoid any kind of information duplication. It often occurs, for example, in the source code and the documentation and fosters inconsistencies as well as errors. **Design patterns** describe proven solutions for typical design problems. For example, the model view controller pattern illustrates how an interactive interface can be implemented in a reusable way by a software structure. Finally, common rules regarding the **programming style** help to achieve consistently formatted source code.

The following recommendations shall ensure the use of common design principles and implementation techniques.

Recommendation	from AC	Explanation
EDI.1: The usual patterns and solution approaches of the selected programming language are used and a set of rules regarding the programming style is consistently applied. The set of rules refers at least to the formatting and commenting.	1	There are often preferred approaches in programming languages to solve specific problems. It is important to use them to achieve an efficient, understandable implementation and to avoid errors. In addition, rules concerning the programming style help to create consistent source code. This increases its understandability, making the software easier to maintain. A few tips on programming styles are given below. It is recommended to stick to an existing set of rules. The source code should have a tidy and consistent layout. This makes it easier to understand and to work with. In particular, the source code can be quickly surveyed and relevant information is easier found. To this end, for example, comment blocks should always be found in the expected position (e.g., before a function definition). In addition, attention should be paid to format functionally similar source code in a similar manner. The name of source code elements (e.g., variables and functions) should already convey important information. To this end, specific words should be used. For example, "DownloadPage" suggests that a network operation to access the specific web page is required. In the case of "GetPage", this additional information is missing. Accordingly, filler words and generic names like "i, j, k, tmp" should rather be avoided. In the case of variables with a short scope, however, the use of these generic names is useful. The key components and modules should be reasonably commented. With the help of the comments, the reader should be able to obtain important additional information (e.g., invariants, constraints, pitfalls) about a module or a component. Thus, commenting on obvious aspects should be avoided. Especially in the case of com-

		ponents, it is helpful to describe their purpose, including how they fit into the remaining software structure. Normally, it should be assumed that developers are the target group who know the programming language and the technical domain. In general, it should be ensured that comments complement other documentation sources and that information is not duplicated.
EDI.2: The software is structured modularly as far as possible. The modules are coupled loosely. I.e., a single module depends as little as possible on other modules.	1	To this end, every module ideally serves a specific purpose. This reduces its complexity at the implementation and interface level. As a result, modules are more understandable, easier to test and to reuse. Finally, this approach helps to ideally keep modules changes locally.
EDI.3: Ideally, there are module tests for every module. The module tests demonstrate their typical use and constraints.	2	Module tests support development and are an important mean to ensure efficient long-term development. Key benefits of module tests include: Module tests provide concrete source code examples that show how to use a module and how to handle errors. Therefore, they represent an important part of the technical documentation. Module tests provide hints about the design quality. Complicated, large module tests indicate that the module may be too complex or too strongly coupled to other modules. This helps to prevent that unfavourable design decisions affect other software areas at an early stage. The ease of automation and the short execution time of module tests help to detect and fix regressions (see the Software Test section) during development. Module tests therefore provide an important basis to perform software structure improvements efficiently (refactoring). The developer ideally creates these functional tests in parallel to the actual module (see recommendation EST.2). Depending on the type of software, however, it is not always possible or appropriate to cover all modules with module tests. Especially in the case of graphical user interfaces, it may be "difficult" to achieve testability at this level.

EDI.4: The implementation reflects the software architecture.	2	I.e., the components defined in the software structure can be found at source code level. For example, there is a specific Java package with the same name for a technical component "persistence". This principle increases clarity, since the entry point can be found at the source code level using the structural diagrams and known terms. In addition, it is possible to avoid casual software structure changes at the architecture level and makes it easier to identify improvements at the architecture level. In general, it is recommended to consistently use terms identified at the level of requirements (see the Requirements Management section) and software architecture (see the Software Architecture section) in the implementation to prevent misunderstandings and errors.
EDI.5: It is continuously paid attention to room for improvement during development. Required changes (refactoring) may be implemented directly or prioritised through the change process.	2	If improvements of the software structure are not performed continuously, the quality of the software structure will get worse. As a consequence, the software is less adaptable and extensible. For example, it may be the case that a function has to be split before an extension can be reasonably implemented. It is recommended to carry out such minor adjustments directly. A good safety net consisting of tests (especially module tests) helps to implement this change quickly and safely. In the case of major changes, it is recommended to analyse the effects in more detail and to implement them prioritised through the change process (see the Change Management section).
EDI.6: Suitability of rules with respect to the programming style is checked regularly. The preferred approaches and design patterns, relevant design principles and rules, as well as rules for permitted and non-permitted language elements may be supplemented.	2	As development progresses, preferred approaches are identified and experience is gained from existing rules. It is therefore recommended to supplement the programming style with these insights and thus prevent errors. For example, it has been recognised that multiple inheritance has proven itself only in the case of interface classes. For this reason, multiple inheritance should only be used for this purpose in the

		future.
EDI.7: Adherence to simple rules concerning the programming style is checked or ensured automatically.	2	In addition to the arrangement of certain rules, it has to be taken care that they are put into practice. In the case of simple rules (e.g., name patterns for variable names), there are often tools that can detect inconsistencies on the basis of the specified programming style (style checker) or directly fix them (source formatter). Depending on the tool type, it is useful to ensure its usage by a common development environment or the build script (see the Automation and Dependency Management section).
EDI.8: Key design principles are defined and communicated.	2	Design principles promote certain work styles during development, which improve the quality of the software structure. For example, the boy scout rule promotes that minor "imperfections" in the source code (also referred to as code smell) are directly addressed when working on a module. By consistently applying this principle, errors are actively prevented. It is recommended to consciously select the key design principles and to communicate them with the development team. They can be embedded in a useful way with the help of the defined programming style.
EDI.9: The source code and the comments contain as little duplicated information as possible. ("Don`t repeat yourself.")	1	The affected places cannot be kept consistent in the long run. Inconsistencies and errors are therefore to be expected in the course of development.
EDI.10: Prefer simple, understandable solutions. ("Keep it simple and stupid.").	1	The goal of this principle is a simple, understandable design. Unnecessarily complex solutions needlessly increase the effort to understand and to extend the software. That does not mean that generic, complex solutions are forbidden per se. You should consciously decide to do it or rather let the solution gradually "grow". This also applies to the use of design patterns. Instead of starting directly with the abstract factory pattern, for example, it may be useful to forego this pattern or rather to use the lightweight facto-

		ry method pattern. Later on, this solution may be further developed (refactoring) to the abstract factory pattern. However, it is important to consciously decide on the basis of the requirements (see the Requirements Management section).
EDI.11: The suitability of the solution, the adherence of the agreed rules regarding the programming style, as well as the relevant constraints regarding the software architecture are systematically checked by code reviews.	3	Some aspects cannot be checked automatically or automation requires substantial effort. These aspects include the understandability and suitability of the implemented solution. Code reviews provide an efficient alternative. In the meantime, available tools can be efficiently integrated into the development process. Typically, one or two experienced developers review a change before it enters the main development branch. This approach allows to detect and fix many errors at an early stage. Code reviews also support the learning process in the development team. It is recommended to embed code reviews via the change process (see the Change Management section).

4.6 Software Test

During **testing**, the software is executed and analysed to find errors. An **error** is a deviation of the actual from the required state. For example, the software does not calculate the product of a number series but its sum (deviation from the functional requirement). In another case, the software provides the result after one second and not after one millisecond as required (deviation from the quality requirement). No proof of correctness can be provided using tests. Rather, testing creates confidence in the software by showing how well it fulfils the desired characteristics. The specific test activities depend heavily on the respective software. Since a complete proof of correctness cannot be carried out in practice, all critical errors must be ideally excluded. It is therefore necessary to determine which aspects of the software are to be tested using which methods and techniques (**test strategy**). The implementation of the test strategy requires considerable effort. For example, a separate test infrastructure or special interfaces may be required. These requirements must be identified at an early stage (see the Requirements Management section) and, if necessary, require decisions and concepts at the architecture level (see the Software Architecture section). This way, situations like the following should be avoided: "The interfaces required for the test are not available. It is too late or too expensive to implement them. As a result, the tests cannot be carried out. The risk of errors during operation increases."

There are several ways to classify a specific test or test case. In accordance to the **test stage**, the following test types can be distinguished:

- **Module tests** (also referred to as unit tests or component tests) show how a specific module works, what restrictions exist, and what constraints must be observed.
- **Integration tests** concentrate on the interaction between certain modules and components. They help to find errors at the interface level.
- **System tests** ensure that the software as a whole meets the specified requirements. These tests are typically performed on software installed in a test environment. In many cases, compliance with the quality requirements can only be checked at this test stage.
- **Acceptance tests** check whether the software meets the requirements from the customer's point of view. These tests are carried out with the participation of the customer and on the basis of the software installed in the target environment. Passing this test level is often the prerequisite for acceptance of the software by the customer.

In practice, it is important to pay attention to appropriate **test automation** and to combine the various test stages effectively with each other. The concept of the test pyramid provides a practical approach. The basic idea is to focus on module tests. These tests have the advantage that they can be easily automated, provide reliable results, do not require a complex test environment and require manageable maintenance efforts. Thus, module tests directly support the development and already find a variety of errors. This is complemented by tests on integration and system test level. These tests are indispensable since only these tests can find errors in the interaction between modules and components.

Another aspect of testing is to gain insight into the quality of the software. This can be quantified using metrics. A **metric** maps a property of the software to a number. In practice, it is thus possible to identify trends and counteract errors. For this purpose, it is important to select and systematically evaluate metrics. For example, the **test coverage** indicates the degree to which the source code is checked by tests. This makes it possible to assess the effectiveness of the test cases.

The following recommendations shall ensure the use of appropriate methods for the early detection and prevention of errors.

Recommendation	from AC	Explanation
EST.1: An overall test strategy is coordinated and defined. It is checked regularly for appropriateness.	2	The test strategy specifies how the testing process for a specific software is designed in principle. It needs, amongst other things, to be considered which test levels are relevant, to what intensity and at what time certain tests must be performed, as well as which test environment and infrastructure is required. It is important to focus on those aspects critical for operation. Quality requirements and constraints form an im-

		portant basis of the testing strategy. For example, if the software shall be used on the operating systems Linux and Windows, tests have to be performed on both operating systems. Therefore it is recommended to develop the test strategy one by one and to build required test infrastructure during development. If necessary, additional interfaces for testing are to be considered conceptually in the software architecture (see the Software Architecture section). The tasks involved are performed prioritised through the change process (see the Change Management section). Finally, it is recommended to embed the resulting test activities in the change process (see the Change Management section). This will ensure that they are systematically performed and that they are visible to all involved in development. Manual steps must be minimised as much as possible to ensure practicability (see the Automation and Dependency Management section).
EST.2: Functional tests are systematically created and executed.	2	Functional tests help to detect errors at the level of functional requirements at an early stage. This requires the interaction of test cases at different test stages. For example, a system test checks whether a user can import a file. In this context, integration and module tests ensure that problems during import are identified and handled properly. Error handling often cannot be checked or is very difficult to check at system test level. Functional tests are often performed very frequently to detect errors during development (also known as regressions). This cannot be done efficiently without adequate automation. Often, there are already suitable test tools. For example, there are xUnit frameworks for many programming languages available which support efficient creation of module and integration tests. On system test level, it may be necessary to create your own test infrastructure.

EST.3: Compliance with the qualitative characteristics is systematically checked.	3	In addition to functionality, aspects such as efficiency or usability often play an important role. These qualitative characteristics are individually defined in every software (see the Requirements Management section) and make a decisive contribution to its acceptance. The key qualitative properties must therefore be checked consistently. Concrete test cases can be determined using quality scenarios (see the Requirements Management section, recommendation EAM.4). The ability to automate the test cases is heavily dependent on the particular quality characteristic. For example, efficiency and reliability can be tested relatively well in an automated manner. However, usability and adaptability are rather poorly automatable. In these cases, manual methods, for example reviews, must be relied on.
EST.4: The basic functions and features of the software are tested in a near-operational environment.	1	This ensures that the software behaves as required in the operational environment. In the case of software with a small scope, it is in principle sufficient to test the main functions manually. However, it is often useful to automate certain partial aspects of the test. This depends on the effort required for automation and how often the main functions are checked. These tests must generally be carried out prior to a release (see the Release Management section). In the case of scientific software, care must be taken to validate the results properly. This can be done through comparison with known solutions or discussion with colleagues. Frequently, errors are not obvious, such as in a simulation result.
EST.5: There is a test for every non-trivial error.	3	This ensures that corrected errors do not occur again (regression). It is recommended to use a test case to trigger the error at the lowest possible test level (module or integration test) and then correct it.
EST.6: There are ideally no non-deterministic functional tests.	3	Non-deterministic functional tests often occur at the "system test" stage. These are tests which do not run predictably and hence can fail although

		the associated test objects, procedures and data have not been changed between testing runs. Non-deterministic tests may indicate an error. The cause of the problem must therefore be determined and fixed. For example, several test cases use a production mail server. For various reasons, the mail server is temporarily unavailable, which often results in failure of these tests. In this case, it makes sense to replace the use of the production server in the test, at least partially, with a test system. In another case, a calculation is carried out through interaction of different threads. Here, unpredictable test results tend to indicate an error in the interaction of the threads.
EST.7: Appropriate metrics are purposefully defined and recorded. The trend of the selected metrics is analysed regularly and potential improvements are identified.	3	There is a variety of metrics for software development. To gain insight into software quality, they must be evaluated on a regular basis. The metrics must therefore be selected deliberately and purposefully. The goal question metric approach provides a practical method for this. In particular, it must be noted that metrics are only meaningful for a specific software. Moreover, it is not the actual measurement that is relevant but its trend. The effects of measures (e.g., the enhancement of test activities) can only be assessed in this way.
EST.8: The trend of the test results, the test coverage, the violations of the programming style, as well as the errors determined by code analysis tools is regularly examined for improvement.	2	Using the **trend of the test results**, it can be seen that certain test cases fail regularly or at times. This can be a sign of a hidden error or an unreliable test environment (see recommendation EST.6). Through the analysis of the test results, the problem is revealed and can be fixed. The **test coverage** indicates which areas of the source code are tested by test cases. Practically relevant are, in particular, the statement and branch coverage. On this basis, conclusions can be drawn about the quality of the existing test cases and the need for improvement. An increasing number of **programming style**

		violations indicates that the consistency of the source code is worsened. This reduces its understandability and contributes to errors. On the basis of the trend analysis, such a tendency can be identified and deliberate counteractions can be taken. **Code analysis tools** analyse the source code for typical error patterns (e.g., comparing a string in Java with "==" instead of "equals"). This helps to find (potential) programming errors. Generally, the tools categorise the results by their severity (e.g., information, warning, error). Depending on the tool and the programming language, false positive results can also occur. Therefore, you should take care to reduce the result list to the practically relevant cases and to configure the tool accordingly. To evaluate these metrics on a regular basis, appropriate tools for their determination must be selected and their execution has to be automated using a build tool (see the Automation and Dependency Management section). Finally, it is recommended to define the point of time of the metrics evaluation as part of the change process (see the Change Management section).
EST.9: The trend of new errors is regularly investigated.	3	For this purpose, errors must be systematically recorded (see the Change Management section). Errors that concern a stable version are relevant for the trend analysis. This enables you to find out how many errors got in a stable software version despite all test activities. By assigning the errors to certain components of the software, areas that are especially susceptible to errors can be identified. On this basis, test activities can be better controlled and their effects determined.
EST.10: The repository ideally contains all artefacts required to test the software.	1	These include, for example, test procedures, test data, and the parameters of test environments. These artefacts are also part of the software and must be kept in a traceable way. In practice, however, there are limits. For example, test records may be too large to be managed efficiently in the repository. In these cases, at least a

		reference to the test record should be stored in the repository. In addition, it should be ensured that the test record is safely stored.

4.7 Release Management

A **release** is a stable version of the software that is distributed to users (e.g., external project partners, colleagues). The **release number** ensures that the release and the associated content are clearly identified. The **release package** contains further files and information in addition to the executable program. This typically includes installation and usage instructions, contact information, an overview of the new features (**release notes**) and the licensing terms.
You define the time of publication and the scope of releases via **release planning**. A ticket system (see the Change Management section) can be used to support this process. It connects the release planning with the change management. As a result, the changes introduced by a release can be easily traced down to the repository.
Depending on the software and development context, various steps have to be carried out until the release can be published. It is recommended to define this process (**release performance**) including the release criteria and to automate essential aspects. This ensures that the release has the desired quality. Here is an example: At the beginning of the release performance, a separate development branch is created to stabilise the software. Therefore, only changes that correct errors or concern the documentation may be applied there. As soon as the required release documentation is available and the software version passes all foreseen tests, the approval criteria for release are fulfilled. Then the release package is created and made available to project partners via the project page. Finally, the underlying software version used for release must be recorded in the repository and the release should be indicated as "completed" in the ticket system.
Finally, here is an important note regarding the distribution of the release package. Before the release package is distributed to third parties outside DLR (e.g., external partners or organisations), the following aspects must be considered:

1. The licensing conditions under which the software is distributed must be defined and accompany the release package. In this context, you have to particularly take care that the obligations and limitations of third-party software are met to avoid legal consequences for DLR.
2. Certain software is subject to export control (e.g., encryption methods). It has to be ensured that the distribution of the release package does not violate existing export restrictions to avoid legal consequences for DLR.

The aspects described have to be considered not only for the special case of the release package. You have to generally comply with them if a software package is distributed to third parties outside DLR.
The following recommendations shall ensure that published versions of the software contain all necessary information and that basic issues are checked prior to release.

Recommendation	from AC	Explanation
ERM.1: Every release has a unique release number. The release number can be used to determine the underlying software version in the repository.	1	The purpose of the release number is to identify the release and the associated content. On this basis, error reports can be clearly associated with the software version in the repository. This simplifies debugging and bug fixing. An example of a commonly used release number format is: X.Y.Z (e.g., 1.0.1). Increasing a particular position in the release number implies statements about the type and scope of the release: Increasing the main release number (X) indicates that major updates are provided by the release. In addition, updating the previous version may not be trivial or changes are incompatible. Increasing the maintenance release number (Y) indicates that a number of new features and bug fixes are provided by the release. Updating the previous version is possible without major difficulties. Increasing the patch release number (Z) indicates that a series of urgent bug fixes are provided by the release. Updating the previous version is possible without major difficulties and is strongly recommended. To find the software version on the basis of a release number, it is recommended to mark every release in the version control system by a tag (see the Change Management section). The tag name should correspond to the release number or be derived directly from it.
ERM.2: The release package contains or references the user documentation. At least, it consists of installation, usage and contact information as well as release notes. In the case of the distribution of the release package to third parties outside DLR, the licensing conditions must be enclosed.	1	This measure ensures that the user has sufficient information about the operation of the software. In addition, in case of questions or problems, the user knows how to contact the developers. The release notes give an overview about the major innovations and improvements of the release. Finally, the licence details define the conditions under which DLR provides the software. The user documentation and the licensing condi-

		tions are part of the software and must therefore be stored in the repository (see the Change Management section).
ERM.3: Releases are published at regular, short intervals.	2	As long as software is actively maintained and extended, releases are ideally published on a regular basis and in short intervals (e.g., every 3 months). This allows to continually receive feedback from users who use the software productively. The direction of development can therefore be better controlled.
ERM.4: The steps required for creating and approving a release are harmonised and documented. The release performance is largely automated.	2	This defines all necessary steps, responsibilities and, in particular, the criteria for approval. The steps for creating a release and the release criteria are part of the change process (see the Change Management section). They should therefore be described in this context. The release performance process can quickly become complex. It is therefore necessary to ensure adequate automation of all essential steps to avoid errors and to reduce effort (see the Automation and Dependency Management section).
ERM.5: The steps required for the creation and short-term approval of a release for critical error corrections are harmonised and documented.	3	If critical errors are reported (e.g., security holes), it may be necessary to publish an unplanned release quickly. It is therefore necessary to consider how the usual procedure can be shortened in these cases to provide bug fixes to users as quickly as possible. The different steps and approval criteria should be described as part of the change process (see the Change Management section).
ERM.6: All foreseen test activities are executed during release performance.	1	Depending on the software and the development context, specific test activities are foreseen to ideally exclude errors on the side of users (see the Software Test section). It is therefore necessary to ensure that these have been performed before the approval of the release. Existing errors and problems are either fixed or at least documented in the release notes before the release is approved.
ERM.7: Prior to the approval of the release, all foreseen tests passed successfully.	2	A release should not contain known errors. If errors are found during the release performance, they must be fixed and their removal should be

		verified by executing the tests again. It is recommended to document the performed tests including information concerning the operational environment, the result and the time.
ERM.8: Prior to the approval of the release, every correction of a critical error is explicitly verified by an independent review.	3	This reduces the chance that certain aspects have been overlooked during bug fixing and that the error has been possibly fixed only partially (e.g., only in one operational environment). In the case of errors that have a critical impact on users, this additional effort is justified.
ERM.9: Prior to distribution of the release package to third parties outside DLR, it must be ensured that a licence is defined, that the licensing terms of used third-party software are met, and that all necessary licence information is included in the release package.	1	Almost every software uses commercial or open source software. By distributing the third-party software as part of the proprietary software, certain conditions have to be met, depending on the licence and the type of use. This may restrict your own licence selection. To avoid legal consequences for DLR due to violation of licensing terms, the stated issues must be checked. It is recommended to determine at an early stage, under which licence (commercial, open source) the proprietary software shall be distributed. As a result, licence compatibility can be taken into account when selecting specific third-party software. As a consequence, the effort for licence checking is limited and you avoid potentially extensive rework. Further information and contact persons with regard to open source usage can be found in the brochure "Use of open source software at DLR".
ERM.10: Prior to distribution of the release package to third parties outside DLR, it has to be ensured that the export control regulations are met.	1	Certain software is subject to export control (e.g., encryption methods). It is therefore necessary to check whether the software is distributed to a partner or external organisation that is covered by a valid sanctions list or is located in a country which is subject to approval. If this is the case, check whether the software contains components relevant for export control. If the result is positive, the resulting obligations and prohibitions must be consistently met to avoid legal consequences for DLR.

ERM.11: Every step of the release performance and its result is logged. All essential artefacts (e.g., release package, test logs) are stored long-term and safely.	3	As a result, there is sufficient information in the long-term to comprehend the release performance and to possibly reproduce the release. In addition, the causes of any errors and problems in the release can be ideally determined. In practice, this recommendation can be ensured by a high level of automation in the release performance (see the Automation and Dependency Management section).

4.8 Automation and Dependency Management

Software development is complex. The software itself is normally already quite large. In addition, several software packages (**dependencies**) in the correct version are required to build the software. Furthermore, various other programs (**development environment**) are necessary to allow developers to create new features efficiently and to ensure their quality. Finally, the software must work under different **operational environments**. It is therefore necessary to ensure this explicitly through tests for each supported operational environment. Required **test environments** must be provided and maintained.

Without the automation of recurring tasks, the complexity described cannot be managed. In particular, the individual steps of the build process must be consistently automated. The **build process** is the process that creates the executable program from the source code and the dependencies (**simple build process**). In the extended sense, this also includes the execution of tests (see the Software Test section) and the creation of the release package (see the Release Management section). In the following, the term "**extended build process**" is used for this process.

There are specialised **build tools** for automating the build process (e.g., Maven, CMake). On this basis, the **build script** is created that automates the build process of a specific software. In many cases, **integrated development environments** (such as Eclipse) enable you to automatically perform the build process "at the push of a button". To do this, they either build on existing build tools or use their own implementation.

In addition, there are different **build variants** that differ in the purpose, the build steps to be performed, and the required runtime environment:

- The **developer build** (also known as private build) creates and checks the software in the local development environment. The developer uses this build variant to check the impact of his/her changes.
- The **integration build** checks the changes of all contributing developers in a neutral test environment. There may be further build steps performed (e.g., special tests, creation of the candidate release package) that are omitted in the developer build for efficiency reasons. The integration build is usually triggered by time or event. The build process must therefore be able to run without manual intervention. An automated build process based on an integrated development environment usually cannot be used for this purpose.

- The **release build** creates the release package on the basis of an approved software version, which can be distributed and is identified by the release number. It represents an extension of the integration build. Additional build steps may be performed that follow up the release creation (e.g., distribution of the release package to the users).

Many steps in software development can only be performed efficiently with the help of an automated build process. In addition, it avoids errors since the participants are relieved from routine tasks. Finally, the build process provides the basis to reproduce achieved development states.
The following recommendations shall ensure the appropriate use of automation techniques to increase efficiency and to deal with dependencies in a structured manner.

Recommendation	from AC	Explanation
EAA.1: The simple build process is basically automated and necessary manual steps are described. In addition, there is sufficient information available about the operational and development environment.	1	An automated build process helps developers to create new functions more efficiently. The build process is normally executed via a simple script call or via an integrated development environment. This reduces complexity, because not every developer needs to know all the details of the programs used and their settings. Complementarily, some additional information is normally required. For example, to use the build script, additional dependencies must be manually installed and their installation directory must be passed by parameter. It is therefore recommended to describe the basic build procedure and, in particular, more detailed information about the development environment as part of the starting guide for developers (see the Change Management section). Finally, the necessary operational environment must also be documented (e.g., as part of the installation instructions, see recommendation ERM.2).
EAA.2: The dependencies to build the software are at least described by name, version number, purpose, licensing terms and reference source.	1	In particular, the licence information provides the basis for assessing the obligations and limitations when distributing the software to third parties outside DLR (see recommendation ERM.9). This documentation obligation is also listed in the brochure "Use of open source software at DLR".
EAA.3: New dependencies are checked for compatibility with the intended licence.	2	The licensing terms of used third-party software can limit, among other things, your licence choice. When selecting third-party software, it is therefore

		important to ensure that its licensing terms are compatible with the intended licence. If no licence has been defined yet, you should take care that the licensing terms impose as few obligations and restrictions as possible (see recommendation ERM.9).
EAA.4: The dependencies are stored long-term and safely.	3	Build tools (e.g., Maven) partly access public repositories to install required third-party software locally and then build the software on this basis. As a result, over time, the problem may arise that a certain version of the software can no longer be reproduced since the required version of a dependency is no longer available. In some cases, it may also be necessary to keep parts of the operational environment. Particularly, it is therefore recommended that you safely store, at the minimum, the dependencies of releases long-term to be able to fulfil possible warranty obligations.
EAA.5: In the build process, the execution of tests, the determination of metrics, the creation of the release package and, if necessary, other steps are performed automatically.	2	The automation of the extended build process is an important basis for efficient development (see recommendation EDI.3) and systematic testing (see the Software Test section). In particular, it forms the prerequisite for the practical implementation of essential aspects of the change process (see the Change Management section) and the release performance (see the Release Management section). It is recommended to optimise the default behaviour of the build process for the developer build because developers are the main target group. For example, on this basis, developers can easily determine that no regressions occur, at least, in the local development environment (see recommendation EST.2). In addition, compliance with agreements, for example, simple rules concerning the programming style (see recommendation EDI.1), can be checked efficiently. In this context, the automated build process provides an easy-to-use interface. Developers do not require detailed knowledge about additional test tools or how to

		access test data.
EAA.6: The build process logs all essential steps and, in particular, enables you to understand the dependencies used during the creation, including their versions.	3	This is an important prerequisite for being able to specifically reproduce a certain software version (see the Release Management section, recommendation ERM.11). For example, in the development environment, the causes of existing errors can be specifically investigated.
EAA.7: Necessary test environments can be provided automatically.	3	Test environments can quickly become quite complex. On the one hand, centralised provision, maintenance and use can lead to a resource bottleneck. On the other hand, unnoticed changes in the test environment can lead to errors. It is therefore useful to automate the provision of the test environment as much as possible. Virtualisation techniques (e.g., docker container) and system configuration tools (e.g., Ansible) form an essential basis. The latter enable you to configure multiple systems automatically. The configuration parameters of the test environment can then be stored in the repository (see recommendation EAA.10) and the test environment can be reproduced on this basis.
EAA.8: An integration build is set up.	2	The integration build allows to regularly check the changes of all developers simultaneously. This makes it possible to detect and fix integration errors at an early stage. The effort to fix the errors is therefore normally lower. A late integration often means that milestones cannot be achieved as planned. To this end, all developers must regularly commit their changes into the repository. On this basis, the integration build can check the software and report the result to the developers. If problems become visible through the integration build, these must be fixed directly. It is recommended to embed this way of working in the change process (see the Change Management section). A prerequisite for an efficient integration build is the automation of the extended build process. All steps must be carried out without manual inter-

		vention and the relevant test environments must be available. For the technical implementation, web-based tools for continuous integration are often used (e.g., Jenkins). These tools can clearly display the build and test status as well as provide functions for the trend analysis of metrics (see the Software Test section, recommendations EST.7 and EST.8).
EAA.9: A release build is set up.	3	The release build consequently automates all significant steps of the release performance (see the Release Management section) to prevent errors as much as possible. This includes, for example, the creation of the release package, the installation of the release in the relevant test environments, the testing of the release, and, if applicable, the distribution and installation of the release in the operational environment (see Continuous Delivery). Depending on the development context, it is important to consider which of these steps can be effectively automated. Prerequisites for an efficient release build are the automation of the extended build process and the availability of the relevant test environments. The integration build is often used as a starting point and further steps are added.
EAA.10: The repository ideally contains all artefacts to perform the build process.	1	This includes, for example, information about the basic build steps and dependencies, the build script, as well as the configuration files of the integrated development environment and the test tools. On this basis, it is possible to restore achieved intermediate versions. In addition, it becomes easier to identify errors that are based on changed settings of the build process.