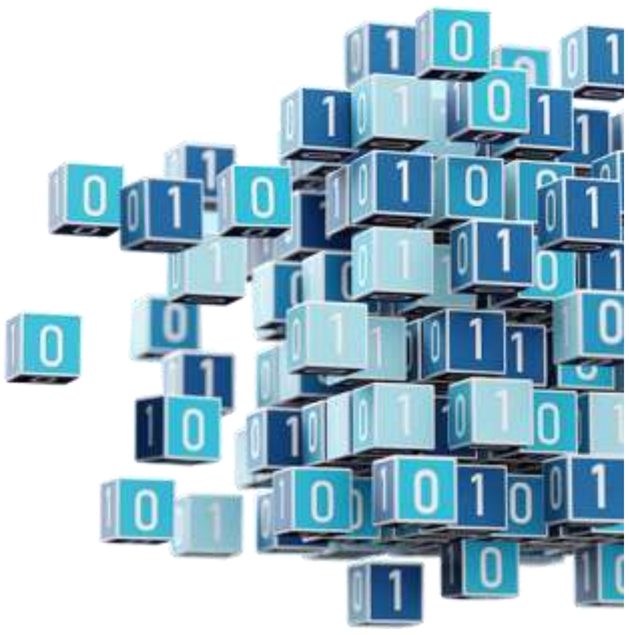
This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 826078.

Privacy preserving federated machine learning and blockchaining for reduced cyber risks in a world of distributed healthcare



Deliverable
D3.4 Manuscript on Lifecycle process

Work Package
WP3 Guidelines, standardization, and certification

Disclaimer

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 826078. Any dissemination of results reflects only the author's view and the European Commission is not responsible for any use that may be made of the information it contains.

Copyright message

Document information

Grant Agreement Number: 826078			**Acronym: FeatureCloud**	
Full title	Privacy preserving federated machine learning and blockchaining for reduced cyber risks in a world of distributed healthcare			
Topic	Toolkit for assessing and reducing cyber risks in hospitals and care centres to protect privacy/data/infrastructures			
Funding scheme	RIA - Research and Innovation action			
Start Date	1 January 2019	**Duration**	60 months	
Project URL	https://featurecloud.eu/			
EU Project Officer	Sevasti SKEVA (Health and Digital Executive Agency (HaDEA)			
Project Coordinator	Jan BAUMBACH, TECHNISCHE UNIVERSITAET MUENCHEN (TUM)			
Deliverable	D3.4 Manuscript on Lifecycle process			
Work Package	WP3 Guidelines, standardization, and certification			
Date of Delivery	**Contractual** 31/05/2021 (M29)		**Actual**	31/05/2021 (M29)
Nature	REPORT	**Dissemination Level**	PUBLIC	
Lead Beneficiary	02 UMR			
Responsible Author(s)	Prof. Dr. Dominik Heider, UMR Dr. Anne-Christin Hauschild, UMR Sabrina Holst, UMR			
Keywords	Software Life Cycle Guideline, knowledge transfer, documentation training			

Table of Content

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 826078.

1 Objectives of the deliverable based on the Description of Action (DoA)

The objective of WP3 is to develop guidelines for a standardized software development process within the academic context and compile a documentation guideline for MDx-ready software (**Objective 1**), which makes conversion from academic projects into MDx software feasible. The goal of these guidelines and recommendations is to ensure that the error rate in the diagnostic process is as low as possible. In particular, as described in **Task 1**, here we developed a straightforward software life cycle process according to IEC 62304 standard. The explicit quality report D3.3 will demonstrate a concrete implementation of such a process for the FeatureCloud project. Moreover, these guidelines will promote further the realization of regulatory requirements, training and enable control of these requirements to ensure product safety. Finally, the guidelines will be made publicly available to provide the same standards for software that will be developed on top of FeatureCloud by third parties.

2 Executive Summary

2.1 Methodology

The requirements relevant for the software life cycle in medical device development (e.g., as defined in IEC 62304 [1] are quite significant and most likely outside of what is feasible for research groups in universities and other research organizations. So far, there is not much guidance on working between the two extremes of not having any software life cycle process at all, and having an exhaustive software life cycle process that complies with all relevant standards. Therefore, we have been studying the corresponding standards and adjusted those requirements towards a software life cycle for the development of software in academia that will potentially be transferred to medical devices. We aimed to embrace most of the benefits and facilitate technology transfer as much as possible while keeping the overhead in a well manageable range – all in the context of universities or similar organizations that engage in research on software in the medical setting.

2.2 Main results

We proposed establishing a limited software life cycle process for research organizations, which has the potential to greatly facilitate and speed up such technology transfer in a controlled and predictable way. Being aware that a complete software life cycle is not feasible for most research organizations, we proposed a subset of elements that we are convinced will provide a significant benefit while keeping the effort in a range that most organizations can easily handle. Our proposal is centered on procedures for software development planning, software requirement analysis, software architectural design, software unit implementation, integration and testing as well as verification and configuration management and problem-solving processes. Depending on the specific needs, the set of elements of a software life cycle process that work best for an organization may differ from what we propose. The fact, however, that a life cycle process is set up at all and that the elements are deliberately chosen is probably a key factor for facilitating technology transfer.

2.3 Progress beyond the state-of-the-art

Medical device software (MDSW) is produced by industrial manufacturers who have to develop compliant to regulatory requirements. Regulatory standards, such as the IEC 62304, focus on establishing industry standards and regulations for MDSW. Recently, the application of computational approaches and artificial intelligence (AI) for purposes of the diagnosis, prognosis, and monitoring of diseases has started a new era in health care and precision medicine. Research organizations spend huge efforts on the development and enhancement of AI guided software tools and algorithms as medical devices to pave the way for a use in clinical practice. However, many obstacles, such as the lack of standardised procedures such as defined by quality management

(QM) and software life cycle (SLC) regulations, hinder a smooth knowledge and technology transfer to industry and day to day clinical application.
However, a complete software life cycle (SLC) as described in these standards is not feasible for most research organizations. Academic research often focuses on developing prototypes and neglects formal documentation or procedures. Moreover, temporary contracts are well established and thus software development is focused on rapidly publishable results, and often happens on a one-person-one-project basis. In order to achieve a reduction of barriers, processes have to be defined and documented according to corresponding guidelines [49].
Here we provide the first set of recommendations for an SCL in academics, lowering the hurdle for many research organizations to set up software life cycle [1,2].

3 Software Life Cycle Guideline

The IEC 62304 defines the software life cycle as a conceptual structure, comprising all steps from the definition of the requirements to the release. It describes processes, tasks, and activities involved in developing a software product and the order and interdependencies between them. Moreover, it defines milestones verifying the completeness of the results to be delivered [1].

Figure 1 shows the software life cycle process and its activities as defined by the IEC 62304. The software maintenance and the development process consist of similar components and are considered essential elements in the software life cycle process defined by the IEC 62304 [1].

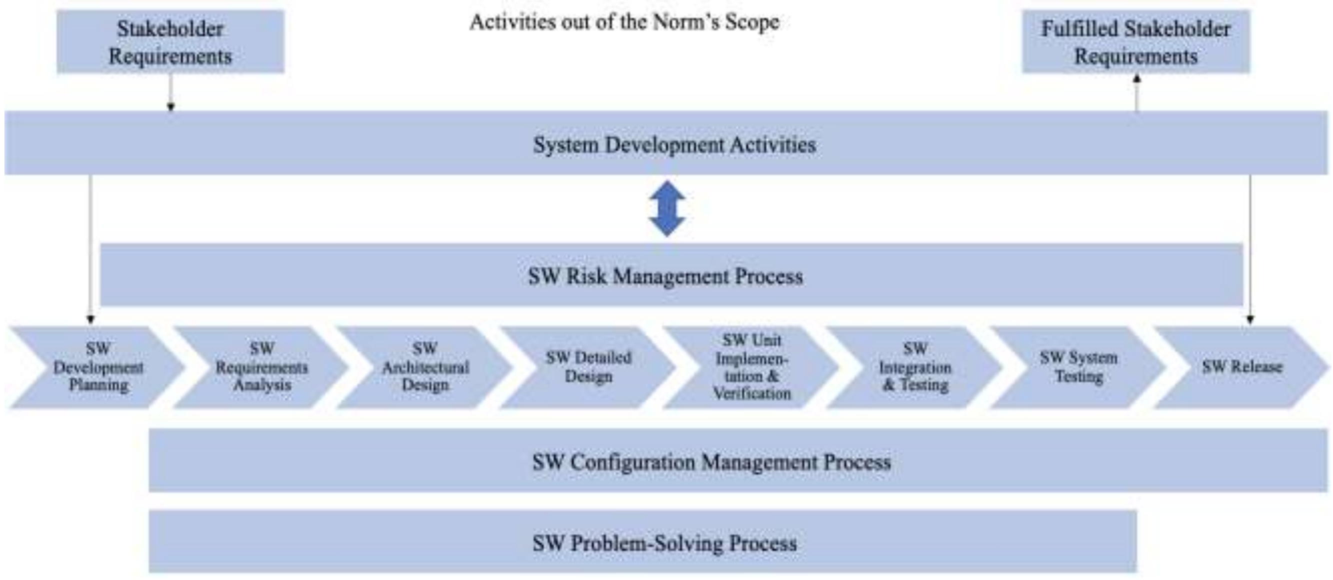

Stakeholder Requirements
Activities out of the Norm's Scope
Fulfilled Stakeholder Requirements
System Development Activities
SW Risk Management Process
SW Development Planning
SW Requirements Analysis
SW Architectural Design
SW Detailed Design
SW Unit Implementation & Verification
SW Integration & Testing
SW System Testing
SW Release
SW Configuration Management Process
SW Problem-Solving Process

Figure 1: Software development process and activities, own representation in the style of ISO [1].

Three essential principles are promoting the safety of medical device software risk management, quality management, and software engineering. Development and maintenance within the risk management process and a quality management system are considered prerequisites for software engineering. The norm identifies two essential additional processes for developing safe medical device software: the software configuration management and the software problem-solving process, as shown in Figure 1 [1]. Those two processes will be covered by the guideline, whereas the software risk management is out of its scope and will be covered in deliverable D3.6. The same applies to the quality management system process, which is covered in D3.2. This guideline focuses on software engineering, including considerations of the planning of the software maintenance process.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 826078.

3.1 Medical Device Software and Regulatory Requirements

The European Medical Devices Regulation (MDR) defines a medical device in Article 2(1) as:

“[...] any instrument, […] software […] or other article intended by the manufacturer to be used, alone or in combination, for human beings for one or more of the following specific medical purposes: diagnosis, prevention, monitoring, prediction, prognosis, treatment or alleviation of disease, […] an injury or disability, an investigation […] or modification of the anatomy or of a physiological or pathological process or state, providing information by means of in vitro examination […].” [2].

Furthermore, the MDR regards software as an active device [2]. The decision if the software is medical device software (MDSW) is referred to as qualification. According to the Medical Device Coordination Group (MDCG), MDSW is software that is intended for one of the medical purposes specified by the MDR or the In Vitro Diagnostic Medical Devices Regulation (IVDR) [3]. MDSW can be an integral part of a medical product or standalone software as an independent medical device [4]. The International Medical Device Regulators Forum (IMDRF) defines software as a medical device (SaMD) as the software used for one or several medical purposes, which are not part of a hardware medical device [5].
Consequently, it is irrelevant for qualification as MDSW whether the software runs on the Cloud, a platform, or a server and whether used by healthcare professionals or laypersons. If the software is part of a medical device, controls a medical device, or is standalone software, it is MDSW. An exception is software exclusively used to regulate the hardware of a medical device that has no medical purpose itself. It is qualified as an accessory for a hardware medical device [3].

In the European Union, the software must have a medical purpose on its own to be qualified as MDSW. The intended purpose of the software, which the manufacturer defines, is relevant for the qualification of the device [3]. In Germany, the software is a medical device if the intended purpose of the manufacturer corresponds to the term “Medizinprodukt” in § 3 of the Medizinproduktegesetz (MPG) [6]. Thereby, the declared application of the product is more important for the qualification than its function. Consequently, the manufacturer must state the application purpose explicitly and consistently to avoid lawsuits. If documentation of vital parameters is the single purpose of the software, it is not a medical device. However, if the application aims to aid doctors in recognizing trends based on the documented data, for example, to determine the proper medication, it is MDSW. Moreover, developers have to adhere to the software life cycle process while developing software [2].

Universities and other research organizations spend enormous effort on developing new software methods and algorithms for use in medical diagnosis and treatment. Once the outcome of this development is mature and robust enough for routine application in treatment or diagnosis, there is a need to bring the knowledge to clinical practice. Therefore, a “smooth” technology transfer to a manufacturer of medical devices would be beneficial, which can be facilitated by adhering to the software life-cycle processes during academic medical software development.

The IEC 62304 is a harmonized standard under the MDR, providing end-to-end guidance for life cycle compliant MDSW development. It does regulate which artifacts and processes are needed but not how the realization has to be conducted [8]. The Food and Drug Administration (FDA) declared the IEC 62304 as a recognized consensus standard for the US [9]. Another norm addressing health software that is not a medical device but deals with healthiness, medicine, and fitness is the IEC 82304. This norm is not harmonized yet and references the IEC 62304 for life cycle conform development [10]. In the following, we will suggest a course of action derived from the IEC 62304 and adjusted to academic capabilities. In case of an intended formal admission and release of the software, strict compliance with the regulatory requirements of the notified bodies has to be ensured as well as the possible need for a clinical evaluation. However, the guideline solely aims to lower the barrier between academia and medical device software manufacturers.

This project has received funding from the European Union’s Horizon 2020 research and innovation programme under grant agreement No 826078.

3.2 Software Safety Classes

The goal of the software safety classes is to control the needed effort for documentation and determine the processes which have to be used during the software maintenance and development process. The IEC 62304 requires different deliverables for each class [1]. This shall not be confused with the four categories I, IIa, IIb, and III for medical devices defined by the MDR or A, B, C by the IVDR, and there is no strict correlation with those classes. On the one hand, a highly critical medical device with class IIb can be classified as safety class A if the software does not affect its criticality. On the other hand, class C software is unlikely to be classified as class I [11].

The classification scheme in Figure 2 shows the safety classes determined by the severity of the resulting harm. If the occurrence of a risk factor is possible, its probability is considered 100%. A software system classifies as:

- A: If it cannot contribute to a hazardous situation or if the hazardous situation, it contributes to, does not lead to an unacceptable risk after considering the external risk control measures.
- B: If the considered risk is unacceptable and the resulting harm is an injury that is not serious.
- C: If the resulting harm is death or severe injury.

Additional risk control measures can be implemented externally to reduce the assigned software safety class. Each sub-software of a product must be classified separately. Unless a separate evaluation is justified, the product will adapt to the highest safety class.

Determining a software safety class in advance is helpful, but should be considered temporary until the software architecture process is completed and all software items are well-defined [1]. Classifying software as A is not recommendable since the MDR demands life cycle conform development, but class A medical device documentation does not suffice the whole life cycle process. However, the norm does not require software architecture, detailed design, implementation, and integration documentation. There are minor differences in the requirements for classes B and C. Therefore, the standard classification of software in the medical context as class C is beneficial. It enables reusing components in another context needing a higher safety class without additional documentation [11]. Consequently, this guideline tries to include all the aspects required for software safety classes C in an appropriate scope for academia.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 826078.

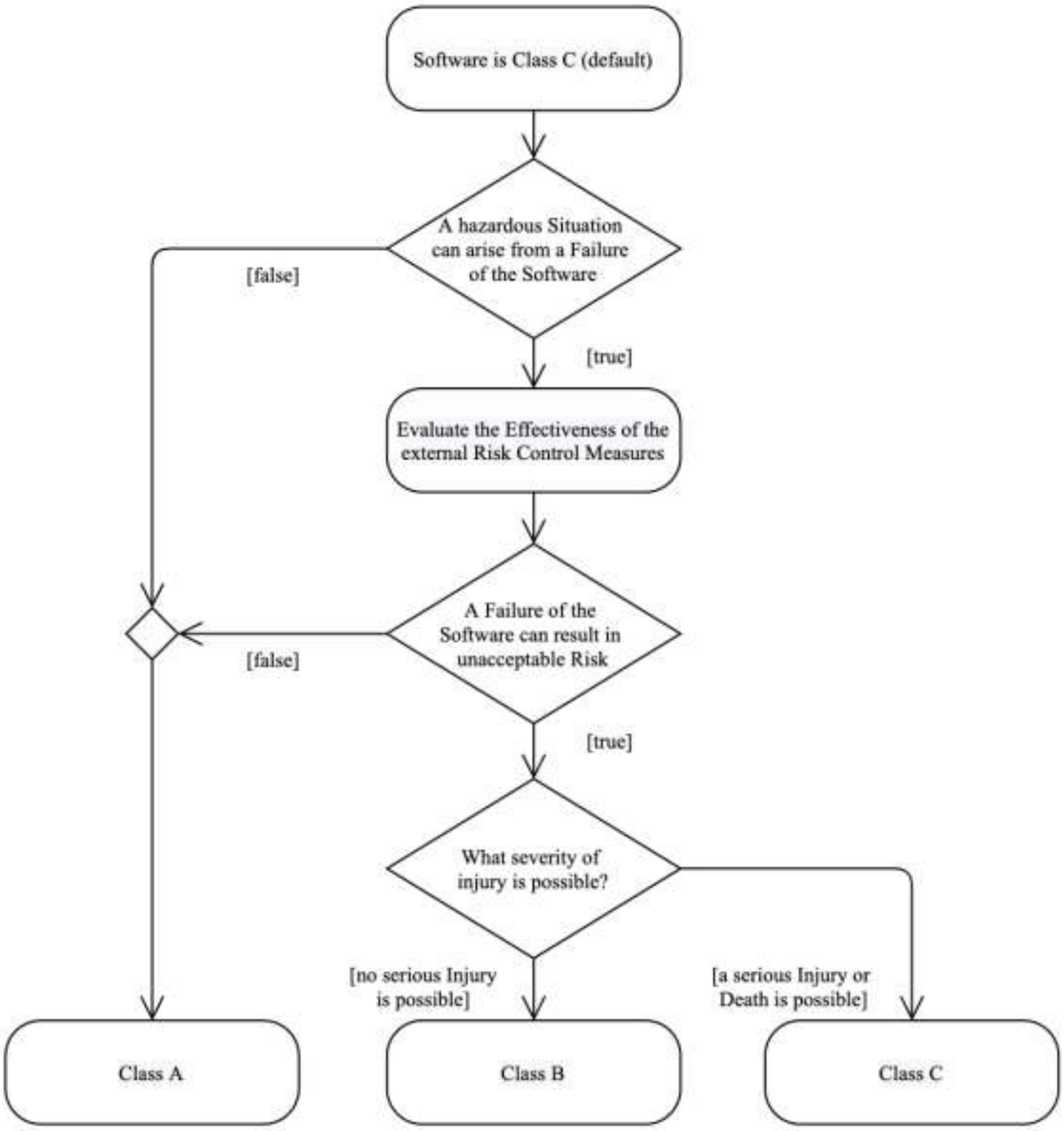

Software is Class C (default)
A hazardous Situation can arise from a Failure of the Software
[false]
[true]
Evaluate the Effectiveness of the external Risk Control Measures
A Failure of the Software can result in unacceptable Risk
[false]
[true]
What severity of injury is possible?
[no serious Injury is possible]
[a serious Injury or Death is possible]
Class A
Class B
Class C

Figure 2: Software safety class determination, own representation in the style of ISO [1].

3.3 Legacy Software

The IEC 62304 defines legacy software as medical device software that was legally placed on the market and is still sold without sufficient proof verifying that it was developed in compliance with the current version of the norm. Instead of walking through all the required stages of the norm for the legacy software, it is sufficient to prove its conformity. The shortcomings to the norm requirements need assessment: the risks of using the legacy software are identified and mitigated, if possible. For this assessment, a risk management process, which is out of this guideline's scope, is essential. A plan has to be drafted in order to close the critical gaps. The manufacturer has to supervise feedback of users, anomalies discovered by the manufacturer, and potential dangers of the legacy software even after its production phase. He has to justify its usage, including positive statements about the safe and reliable performance of the legacy software, and document its version [1].

In academia, it is essential to document the usage of legacy software. In general, legacy software has to fulfill the requirements of the IEC as well, but the scope of action only demands to close gaps

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 826078.

if it reduces the risk of usage. However, there is a difference if legacy software is refined or only used within academia. All above need consideration when planning the documentation.

3.4 Software Development Process

As mentioned above, the software life cycle guideline defines the software development process as a set of activities. In the following, we systematically introduce the most vital activities described by IEC 62304. Moreover, we will highlight supporting theoretical background information, frameworks, and advice for suitable applications.

3.4.1 Software Development Planning

The first component demanded by the IEC 62304 is a software development plan anticipating all the activities of the software development process. Essentially, the entire life cycle model has to be defined or referenced. The development plan is updated regularly during the project and includes the following:

- The used process and the deliverable results of the activities and exercises
- The traceability between requirements, software system testing, and implemented risk control measures
- The software configuration and change management
- The problem-solving process.

Moreover, it addresses norms, methods, and tools used to develop software items and verifies the configuration elements, including software of unknown provenance (SOUP). SOUPs have been created beforehand without the intention to use them in a medical device and are commonly available, such as base packages of a software environment.
Each document produced during software development contains the title, purpose, and responsible person [1].
While the requirements of the standards are complex, we will highlight key components that are essential considering the scope of academia. Therefore, it is beneficial to differentiate between: the process description and the development plan.

The process description is a development standard operating procedure (SOP) providing structured solutions for repeated application problems [12]. The MDR demands the developer to use a quality management system, including the product realization and development process [2], implemented through the SOP. It is a general handbook defining which activity will be completed, when, by whom, how, and with which input and output. It is independent of a product or project and can be reused in another context [13]. For this reason, developing SOPs that can be applied in many projects is a good approach for academia.

Figure 3 shows a turtle diagram, a process analyzing tool that can identify the process elements and define an appropriate process [14].

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 826078.

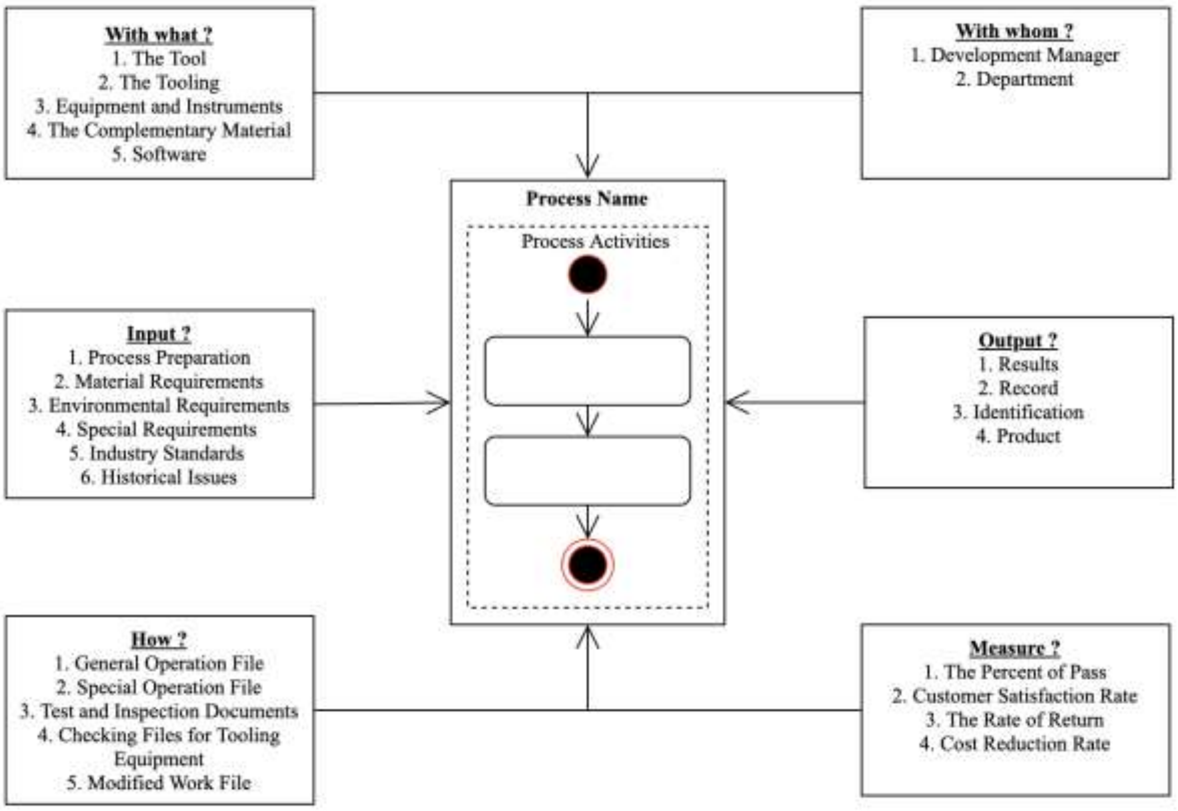


Figure 3: Turtle diagram, own representation in the style of Guo H, Zhang R, Chen X et al. [14].

Since the life cycle model has to be defined or referenced by the development plan, a software engineering model must be chosen to provide a general structure for the development phase. Figure 4 shows an example for a waterfall model including the required software processes of the IEC 62304.

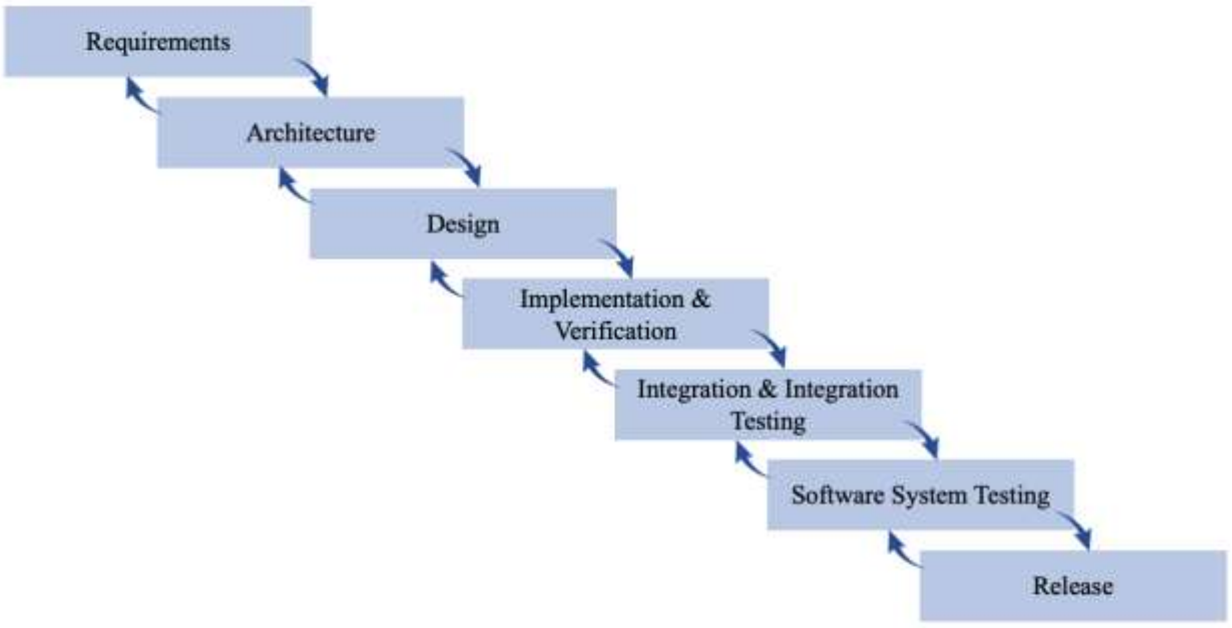

Requirements
Architecture
Design
Implementation & Verification
Integration & Integration Testing
Software System Testing
Release

Figure 4: IEC 62304 software development activities within a waterfall approach, own representation.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 826078.

The developer team is free to choose a model that best fits the characteristics of the project. For instance, a commonly used approach in the medical device development context is the V-Model. It fits regulatory requirements and guides through the process of producing the required deliverables to achieve regulatory conformance [15].

The development plan describes the concrete product or project and the process description. It defines concrete milestones to corresponding deadlines, assigns concrete staff to the pre-defined roles, and refines measures or tools adjusted to the specific project [13]. It is challenging to meet the regulatory requirement of defining tools, testing, and configuration management in advance. Their definition often depends on external factors and restrictions that are not predictable or develop only during the development process. Therefore, the development plan is updated over the project period, while the pre-defined process description stays the same. Through the provision of those two documents, the development process gets more transparent and well-structured.

Additionally, the norm suggests a definition of a coding guideline for the implementation of the detailed design (section 3.4.4) and its verification (section 3.4.5), to make the source code more understandable, and increase its quality by reducing mistakes. Naming conventions like capitalization or usage of underscores can be defined as well as indentations and use of brackets [21]. Moreover, the processes can require the specification of software documentation or nomenclature for classes or functions, for instance. Some programming languages like Java already have coding conventions that can be adapted for the coding guideline. The definition of a coding guideline is advisable in academia and is part of the development planning. Even in one-developer projects, a coding guideline is helpful in case of project transfer or for verification.

3.4.2 Software Requirements Analysis

A requirement is a condition or capability the user needs. This can be the capability to solve a problem or achieve an objective that must be met. Additionally, it can be a condition possessed by a system or a system item [22]. We distinguish between the customer needs, design input, and software requirements. Design inputs translate customer needs to formal documented medical device requirements. Software requirements formally specify what the software does to fulfill the design inputs and customer needs.

The regulatory standards demand to derive and document the software and system requirements, which are equivalent in software as medical device. Additionally, the norm includes functional and performance requirements with physical characteristics, the computing environment, inputs, and outputs, including data characteristics, such as value range, limits, and typical values. Equally important are the interfaces of the software considering the user requirements and the technical system and data interfaces, including the requirements to the database, data security, and compatibility considerations. Moreover, there are installation, maintenance and operation, regulatory and network requirements. The requirements have to be updated and verified [1,53,54].

As mentioned in 3.4.1, high-level requirements have to be defined in advance. It is vital to gather the stakeholder demands first and then derive the technical product requirements [23]. One distinguishes three types of requirements: functional, non-functional, and constraints [24]:

1. A **functional requirement** specifies a function a system or the component of a system must be able to perform [22]. Examples are functions, process descriptions, and scenarios defining a system’s reaction to specific inputs. Data and interface requirements to deploy the system in a particular environment, as well as workflows and business processes, belong to the functional requirements as well.
2. A **non-functional requirement** describes a qualitative feature a system or a system component must-have. Examples are reliability, availability, maintainability, functional safety, and information security.

3. **Constraints** are requirements restricting the modality of how a system can be realized. Laws, costs, and business processes, as well as limitations through infrastructure, can be mentioned exemplarily [24].

The structure of the requirement document is adjusted to the project. There are different approaches to structure the requirements. One can differentiate between constraints, functional, and non-functional requirements or describe the project as a black box, specifying the software performance via its interfaces. For instance, the user interface, technical interface, and runtime environment can be used as a chapter structure [25].

The high-level requirements can be defined in a requirement specification, including the desired properties of the performance of an artifact. This document can be seen as a goal that has to be achieved under certain restrictions [26]. Within the specification, requirements can be documented in natural language as well as in graphic notation like Unified Modeling Language (UML) diagrams. Both techniques have advantages and disadvantages. On the one hand, natural language is not as clear and exact as graphic notation and can be misinterpreted. On the other hand, natural language is faster and easier to use and does not require background knowledge like the rules of a formal modeling language, which especially many of the stakeholders do not have.
A possible requirements template for user stories is: As a <user role> I want <the action> in order to/ because <benefit>. A system requirement can be specified like this: Given <prerequisite> and <possible additional prerequisites> when <action> then <result/ postcondition> [27]. The requirements specification has to be under configuration management or version control (see Section 3.5), which means that changes are handled systematically and that the system integrity is maintained over the whole development process [23]. It is beneficial to start with more general requirements within the specification, mainly using natural language and refining them later using graphic notation. Requirements include acceptance criteria needed for testing and verification.
Consequently, it is important that the original requirement within the specification is always referenced as well as software architecture or design, in order to establish “traceability”. A trace can be defined as the bidirectional link between a source and a target artifact [50]. Establishing and using traces is referred to as traceability [51]. “Requirements traceability refers to the ability to describe and follow the life of a requirement, in both a forward and backward direction (i.e., from its origins, through its development and specification, to its subsequent deployment and use, and through all periods of ongoing refinement and iteration in any of these phases).” [52]. This means that traceability implies the comprehension of a design starting with the source of a requirement, its implementation, testing, and maintenance and helps ensure the developed software is of high quality [51]. High quality and safety are critical concerns in the medical device domain; therefore, traceability must be implemented within the entire development process. The configuration management process completes this implementation by preserving the created links making changes visible and transparent. Furthermore, one should keep in mind that requirements are functionalities. The detailed design document defines the implementation.

There are different types of requirements elicitation techniques. The most important factors to find the appropriate requirement analysis technique are the chances and risks within a project, especially the characteristics and nature of the stakeholders. Moreover, the expertise of the requirements engineer with the techniques are as well relevant [28]. The Kano model distinguishes must-have requirements, one-dimensional requirements, and attractive requirements, determining the appropriateness of the requirement elicitation techniques. The most popular ones are interview techniques, observation techniques, system-or product archeology, simulation and animation techniques, and creativity techniques [29].

The interoperability is analyzed to define data or technical interface requirements. The IEEE defines interoperability as “the ability of two or more systems or components to exchange information and to use the information that has been exchanged” [22]. Table 1 shows the differentiation between structural, syntactic, semantic, and organizational interoperability, as well as typical standards. The

This project has received funding from the European Union’s Horizon 2020 research and innovation programme under grant agreement No 826078.

structural layer ensures that the data stream can get from one system to another. The syntactical layer identifies information units within a data stream, and within the semantic layer, one ensures they have a consistent meaning within the whole system. Within the fourth layer, mutual role definitions, workflows, and authorizations are defined [30].

Table 1: The model of layers of interoperability, own representation in the style of Johner [30].

Level	*Task*	*Typical Standards*
Organisatorial	*Enable system wide processes, roles, and permissions*	*BPMN, IHE*
Semantic	*Obtain a uniform understanding of information units*	*Classification systems such as ICD, LONIC, ATC, value tables in HL7 and DICOM*
Syntactic	*Recognize information units in the data*	*XML, CSV, HL7, DICOM, ...*
Structural	*Transfer data from one system to another*	*Protocols of the OSI layer model such as TCP/IP, FTP, HTTP etc., data and network protocols*

In order to evaluate if the software requirements are complete, the product quality model of the ISO/IEC 25010 can be referenced, as shown in Figure 5. It categorizes the product quality properties into eight characteristics: Functional suitability, reliability, performance efficiency, usability, security, compatibility, maintainability, and portability [31]. Besides, institutions like the Johner Institut provide checklists online to evaluate the requirement's integrity [53,54].

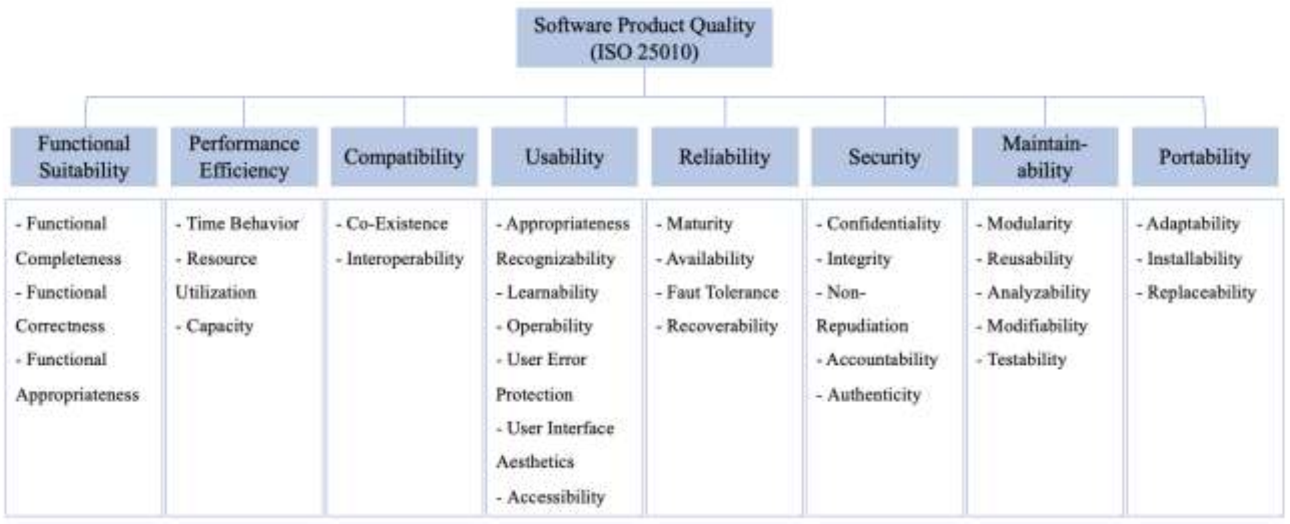


Figure 5: The product quality properties by ISO 25010, own representation.

Moreover, the norm demands the definition of requirements for maintenance. In academia, that might seem obsolete since the maintenance phase starts after the product delivery, and usually, the focus of academia is on developing and prototyping new methodologies. Nevertheless, maintenance has to be considered at the beginning of the software life cycle to ensure post-delivery support is possible [32]. Therefore, in academia, a maintenance plan does not have to be complete, but has to define

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 826078.

all factors influencing the development or architecture. Examples are contact forms for user feedback or the need for log files, including all recognized errors.

The MDR requires post-market surveillance to collect and review the gained experience from the device, which has to be facilitated [2]. Responsibilities for the modules have to be defined. Third-party libraries like software packages used in the project should be listed to enable future observation of their error reports. Even if they do not have a medical device background, listing them is of high importance since they are developed and maintained by someone else, which has to be observed as required by the development planning for SOUP as well as the post-market surveillance by the MDR. In general, one should ensure that the software can be maintained in the future.

3.4.3 Software Architectural and Design

Regulatory standards, such as the IEC 62304, demand the definition of essential structural software components, identification of their primary responsibilities, visible features, and their interrelations. The architecture describes the software structure implementing the software requirements. Internal and external interfaces have to be documented as well as functional, performance and hardware requirements of SOUP. Moreover, the architecture has to be verified. The system has to be divided until it is represented through software units, which are sets of procedures or functions encapsulated in a package or class that can not be divided further. Each software unit and interface needs a verified detailed design to ensure correct implementation [1]. A software architecture consists of the structure of the system in combination with architecture characteristics the system has to support, architecture decisions, and design principles. The structure is the architectural style the system is implemented in. Examples are layers or microservices. Architecture characteristics define the success criteria of a system which are required so that the system works properly. Examples are availability, scalability, or security. Architecture decisions define rules for the system’s construction, formulating constraints. Due to architecture decisions, the developers know what they may or may not do, for example, who is allowed to access a database. A design principle is rather a guideline than a hard rule implementing the software architecture. It is more detailed than an architecture decision that could never cover every option or state of all system components. A design principle could demand asynchronous messaging, providing guidance for the preferred method and helping the development team choose the appropriate communication protocol [33].

Architecture categorizes into monolithic and distributed architecture types. In monolithic approaches, the code is deployed within a single package, whereas in distributed architectures, subsystems are physically separated, exchanging resources through standard interfaces [33,34]. Typical monolithic approaches are layered architecture, pipeline architecture, and microkernel architecture. The service-based architecture, event-driven architecture, space-based architecture, service-oriented architecture, and microservices architecture are distributed approaches [33]. The norm does not require a particular architecture, but the software structure has to be defined within a framework chosen by the development team. While describing this structure, cohesion and coupling are general terms that have to be considered in software architecture design. Cohesion refers to the dependencies within a subsystem, while coupling refers to dependencies between subsystems. Good software should have low coupling and high cohesion to be well testable and maintainable [35].

In order to reach a low coupling, only a few methods or variables should be defined as public, the amount of transfer parameters has to be minimized, inheritance over the borders between software items should be avoided, and methods should be abstracted through interfaces [36]. UML is a widely accepted notation standard enabling a consistent, precise notation to describe software items and their relations. It facilitates a common understanding of the architecture for all developers and stakeholders [37]. Therefore, usage of UML as a notation standard is highly recommendable. Class and activity diagrams are especially useful in academia since they have simple notation rules formalizing architectural decisions.

This project has received funding from the European Union’s Horizon 2020 research and innovation programme under grant agreement No 826078.

At least two documents should be provided in academia: the software architecture description and the detailed design. The architecture conceptually defines data storage, interfaces, and logical servers. The modularization is described within the detailed design, conceptualizing the architecture as a rough concept and how the requirements are explicitly implemented in the software. While the most important classes should be mentioned, it is neither necessary nor achievable in academia to conceptualize the exact codebase. Moreover, the graphical user interface (GUI) is critical for a system. To most users, it represents the system because it is what is seen and touched. Therefore an interface that makes using the program accessible, userfriendly, and productive has to be provided [38]. Consequently, the user interface and its static and dynamic performance should be described. A mock-up is useful to give a concrete definition of the user interface, and activity diagrams can be used to specify the order of the screens.

3.4.4 Software Unit Implementation, Integration, and Software System Testing

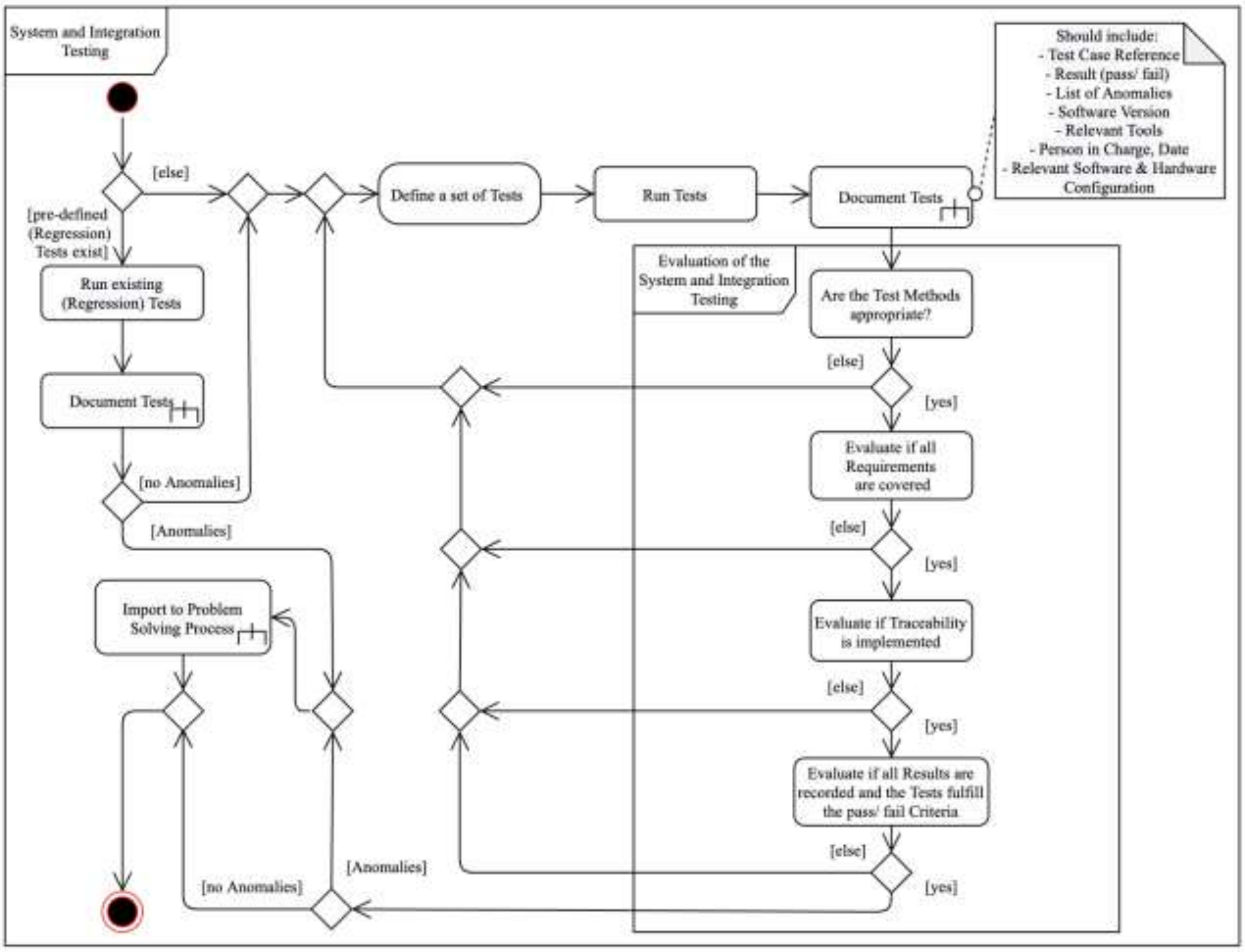

System and Integration Testing
[else]
[pre-defined (Regression) Tests exist]
Run existing (Regression) Tests
Document Tests
[no Anomalies]
[Anomalies]
Import to Problem Solving Process
[no Anomalies]
[Anomalies]
Define a set of Tests
Run Tests
Document Tests
Should include:
- Test Case Reference
- Result (pass/ fail)
- List of Anomalies
- Software Version
- Relevant Tools
- Person in Charge, Date
- Relevant Software & Hardware Configuration
Evaluation of the System and Integration Testing
Are the Test Methods appropriate?
[else]
[yes]
Evaluate if all Requirements are covered
[else]
[yes]
Evaluate if Traceability is implemented
[else]
[yes]
Evaluate if all Results are recorded and the Tests fulfill the pass/ fail Criteria
[else]
[yes]

Figure 6: Activity diagram system and integration testing workflow required by the IEC 62304, own representation.

Each software unit must be implemented and verified; the verification has to be documented [1]. The IEEE defines implementation as “The process of translating a design into hardware components, software components, or both.” [22]. The detailed design translates into source code. At this point, the resolution of the specification ends, and the composition of the executable software begins. We recommend using a coding guideline specifying a consistent coding style. Every unit is verified to make sure it works as defined in the detailed design and follows the coding guideline. Afterward, each software unit has to be integrated, verified, and tested independently and dependently [1].

This project has received funding from the European Union’s Horizon 2020 research and innovation programme under grant agreement No 826078.

Integration is defined as “The process of combining software components, hardware components, or both into an overall system” [22]. The integration strategy derives from the software architecture, which determines the order of integration of software units. The software units affecting safety require more detailed testing. The IEC 62304 allows the combination of integration testing and software system testing with a common set of activities but requires covering all software requirements. They may be tested in earlier stages before system testing and can be combined in case of interdependencies. The sufficiency of the testing procedure, the verification, and the integration strategy for all software requirements have to be evaluated appropriately, traceability between requirements. Their verification and testing are recorded, and the results fulfill the pass or fail criteria. The testing has to be repeated and may be adjusted in case of changes. The testing and the responsible tester have to be documented to ensure it is repeatable. After the integration of software units, regression tests have to be performed to show that no anomalies are imported into the software. Regression tests should be automated and are defined as tests that are necessary to evaluate that the change of a system unit does neither affect its functionality, reliability, or performance nor causes additional faults. Anomalies found in the system and integration testing have to be imported into the problem-solving process, see Section 3.5 [1]. Figure 6 shows an activity diagram modeling an IEC compliant system and integration testing approach.

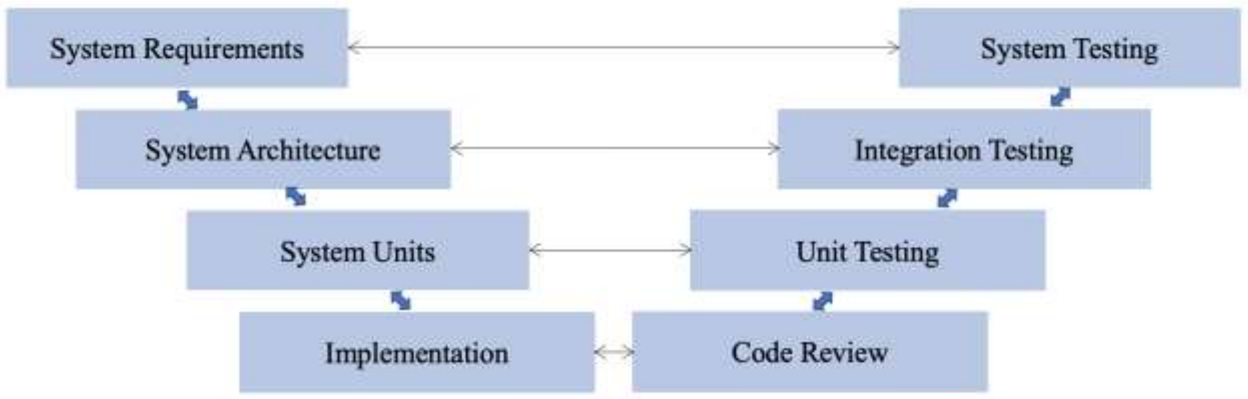


Figure 7: Software testing at certain levels within a V-approach, own representation.

Different tests are appropriate to test or verify various artifacts required by the IEC 62304, shown in Figure 7. Software testing usually takes place at certain levels within a software’s life cycle. Unit testing, integration testing, and system testing are differentiated. Unit testing means verification of the functioning of an element of the software that can be tested separately. The tests are conducted in isolation. Usually, the code is tested through the responsible programmer, using debugging tools for support. In order to evaluate the interaction between software items, integration tests are performed. There are different test strategies, like top-down or bottom-up. They are used during several stages within the development process and are tailored to each integration level [32]. Integration tests are performed to test the data transfer, ensuring that all internal and external interfaces of the software work correctly. Program behavior at the extremes of the input and result ranges, as well as responses to specific, invalid, or unexpected inputs, are evaluated [1]). Finally, the behavior of an entire system is tested through system tests. They focus on functional and non-functional requirements like speed, security, or performance. Furthermore, external interfaces and hardware devices are evaluated as well [32].
While unit tests are often defined as tests of classes and functions, the norm states software units are integral parts of the software architecture. It is therefore critical to test the software units specified within the software architecture[39].

A system is tested to show if a specific functionality exists, verifying the functionality and performance of a program with respect to the system requirements [1]. A test is successful, if it passes the acceptance criteria which are defined in the requirements specification, the interface design of the

This project has received funding from the European Union’s Horizon 2020 research and innovation programme under grant agreement No 826078.

software development plan, and the coding guideline [21]. Typical types of black-box testing are equivalence class partitioning, boundary value analysis, model checking, and random testing [40]. White box tests, which rely on specific knowledge of the software's internal structure to evaluate the test results and find mistakes within the implementation, can be used as well [1]. White box testing is also referred to as structural testing [41], and common approaches are loop, basis path, branch, control- and data flow testing [42]. Test results can be stored in a log document and should be under versioning control.

Non-automated testing by humans is possible as well. Human testing techniques are less formal than automated testing and should be conducted between program coding and the beginning of computer-based testing. The three fundamental human testing methods are code inspections, walkthroughs, and usability testing. In a walkthrough, a small group of people, including only one programmer involved in writing the code, reviews the code. Code inspection contains a set of techniques for error detection, and the usage of a guideline to check typical sources of bugs is recommendable. Both techniques are very similar, but within a walkthrough, the participants theoretically execute each test case and question the programmer instead of simply reading the program or using an error checklist [43]. For usability testing, practical exercises which have to be conducted by the user and contain all aspects of the software are developed, and afterward, the users give their feedback [44]. In alpha and beta tests, the software is released to the actual end-user. Within the alpha testing phase, selected users report bugs, followed by the beta phase, where the software is tested by a larger and more representative set of users [45]. Consequently, beta tests are used to verify that a computer system satisfies its requirements, and they are performed before the software is released [46]. Providing medical software to potential users for testing purposes is difficult because placing medical devices on the market, without a CE Mark proving its conformity with European laws, is not allowed [47]. Thus, official beta tests should be avoided since they are not allowed for medical device software. If user feedback is needed, it must be explicitly emphasized that the software is not approved and only provided for testing purposes. In this case, relevant bodies should be contacted for approval. Moreover, the norm does not require code reviews but suggests the definition of a coding guideline to verify the implementation. Code reviews can be proposed as a non-automated technique to find bugs in the source code and verify its compliance with the coding guideline. The results should be documented in a provided standard form sheet or tools like Team Foundation Server or MedPack [55].

3.4.5 Verification

The regulatory standards such as the IEC 62304 define verification as a confirmation that all specified requirements are fulfilled by providing objective proof. It is required in several stages of the software development: the software verification has to be planned (section 3.4.1), and software requirements (section 3.4.2), software architecture, detailed software design (section 3.4.3), software units as well as their integration (section 3.4.4), changes and the solutions of software problems (section 3.5) have to be verified [1].

In order to validate the software requirements, one examines if they are coherent with the system requirements. The software requirements must be consistent, may not contradict each other, unambiguous, clearly identifiable, and traceable to the system requirements or other sources. The definition of testing criteria based on the requirements has to be realizable. For the verification of the entire architecture, all underlying system and software requirements have to be implemented. It has to support the interfaces as well as SOUP items. Detailed design is verified whether it implements and does not contradict the software architecture. For verification of software units, four acceptance criteria can be defined: the requirements, the interface design, the detailed design, and the coding guideline. Each requirement is tested through a test case. In case the test is passed, the requirement is fulfilled. The code may neither contradict the interface design of the software unit or its detailed design nor the defined coding norms.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 826078.

The verification of all architectural, system and software requirements has to be documented. For verification of the software integration, it has to be shown and documented that the integration of a software unit is realized according to an integration plan derived from the software architecture. Integrated software items have to work correctly to ensure the whole software operates safely. Consequently, the testing of the software system is the verification of the functionality of the software. Changes in the verified software system require a repeated verification. For this reason, system tests have to be modified or repeated. If a problem caused the change, its solution has to be verified. The verification determines if the problem is solved. The corresponding problem report is closed if the change requests are implemented and no additional issues are caused [1]. Overall, verification is an activity of high importance throughout the whole development process. In order to verify more mature artifacts, one has to verify its foundation as well. This hierarchy should always be kept in mind.

3.4.6 Software Release

Before software release, the software verification needs to be complete, and the results evaluated [1].
This means:

- All known remaining issues have to be documented and assessed.
- Documentation of the procedure and the software development environment have to be recorded as well as the version of the released software.
- All activities and tasks of the software development plan must be completed and documented.
- The medical device software, all configuration elements, and the documentation have to be filed for the whole lifetime of the software defined by the development team.
- Procedures to ensure a reliable delivery without any damaging or unauthorized adjustments have to be defined.

After the software is released, all changes and updates are implemented within the software maintenance process, following the same steps as during the software development process. The concrete requirements of the norm concerning this process are out of the scope of this guideline, since the focus is on software development within academia. The post-delivery maintenance decisions which have to be made are included within the software development planning in section 3.4.1.

The software release does not include the release of the software as a product. It is not equal to the market release and only means that the software is finished from the developer's point of view. The market release depends on the target market and the demands by the responsible authorities [48]. If the development process is well defined and followed, and the previous sections of this guideline are considered, the complete verification, as well as the required documents, are delivered by default. Well-implemented traceability is essential to ensure the development process can be archived transparently.

3.5 Configuration Management Process and Problem-Resolution and Handling Process

This process has to be implemented for the entire software life cycle, including technical and administrative procedures to identify and define configuration items and SOUP as well as their documentation within a system. Change requests, changes, and releases are controlled and documented.
This is important in case an item has to be restored and to determine its components, and provide the history of its changes. Change control demands that configuration items can only be changed if the associated change request is approved and the changes have to be verified again. Consequently, unauthorized and unintended changes can be prevented. Change requests, their approval as well as the relevant problem report have to be filed to ensure they are fully implemented [1]. Since changes have to be permitted, the person in charge of change management should be

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 826078.

defined in advance as well as clear criteria for the permission of a change request. The usage of a ticketing system is recommendable. All documents should be under versioning control to facilitate tracing of changes. Riemenschneider et al. suggested hiring permanent technical academic staff supporting the software development of Ph.D. students and postdocs [49]. In case such a quality manager is available, they could be in charge of change control or support the correct implementation of the necessary processes.

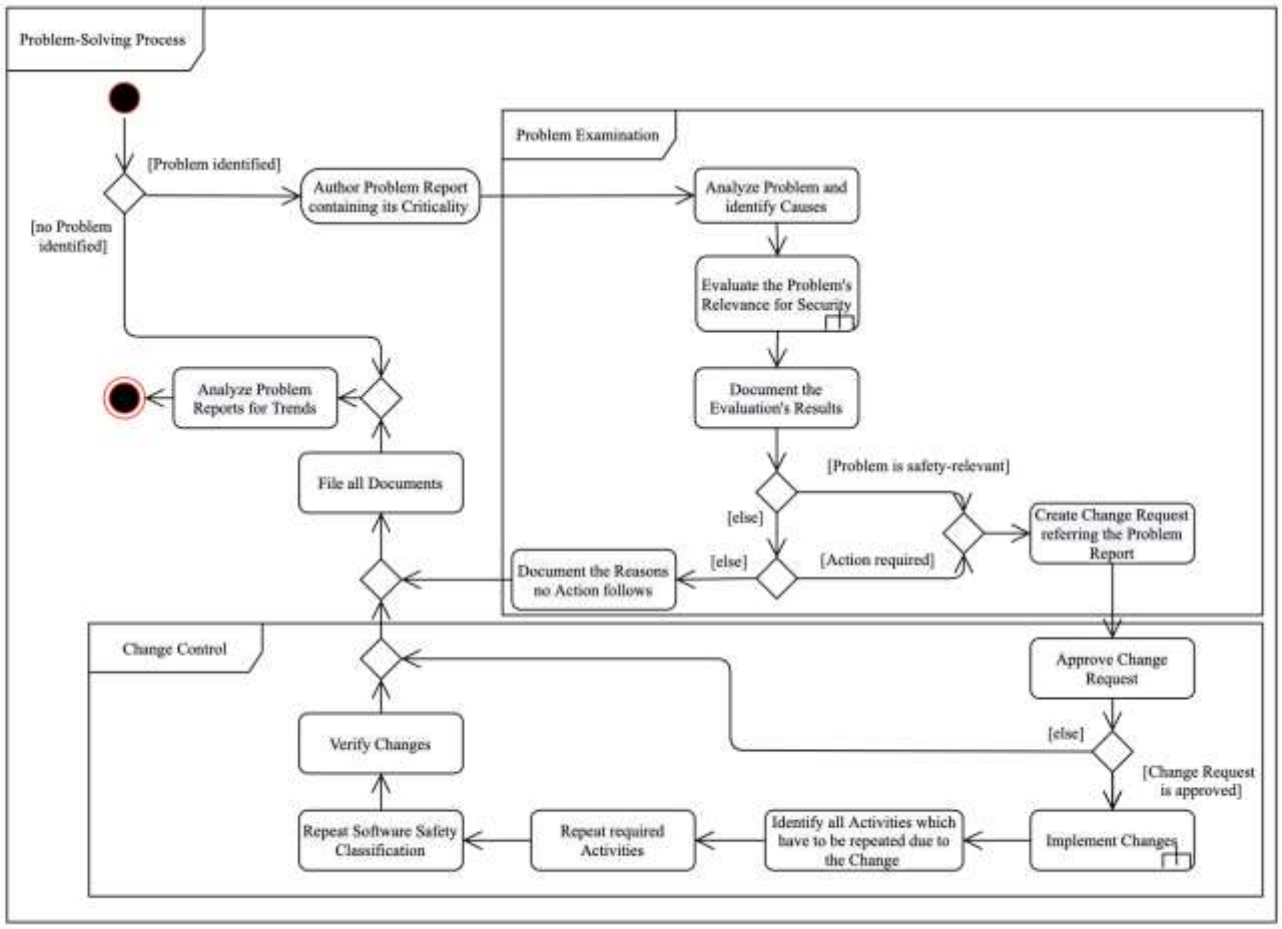

Problem-Solving Process
[Problem identified]
[no Problem identified]
Author Problem Report containing its Criticality
Problem Examination
Analyze Problem and identify Causes
Evaluate the Problem's Relevance for Security
Document the Evaluation's Results
[Problem is safety-relevant]
[else]
[Action required]
[else]
Create Change Request referring the Problem Report
Document the Reasons no Action follows
Analyze Problem Reports for Trends
File all Documents
Change Control
Approve Change Request
[else]
[Change Request is approved]
Verify Changes
Repeat Software Safety Classification
Repeat required Activities
Identify all Activities which have to be repeated due to the Change
Implement Changes

Figure 8: Activity diagram problem-solving process required by the IEC 62304, own representation.

The regulatory standards such as the IEC 62304 demand problem-solving processes if a problem or nonconformity is identified within the development process to analyze and solve it. That refers to anomalies found within the software integration or integration- and software system testing. The problem-solving process is defined during the project planning phase. A problem report has to be created for each problem that occurred, including its criticality. The problem has to be analyzed. Its safety relevance is evaluated using the software risk management process. The norm does not require the correction of issues that are not safety-relevant. In case correction measures are necessary, a change request has to be created. Otherwise, it is documented that no action follows. The implementation of those solutions is bound to the change control within the configuration management process and has to be verified. Moreover, all documents have to be filed as specified by the life cycle process, and the problem reports have to be analyzed for trends [1]. Figure 8 depicts the problem-solving process, as well as its connection to the change control.

Configuration management, including change management and a problem-solving process, should be defined at the beginning of the project within the software development planning (section 3.4.1).

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 826078.

4 Software Life Cycle Process Check-List

Based on the analysis performed in this task and deliverable, the following list shows the key steps that are essential to develop a software life cycle process for an academic group or institution:

1. Describe the process and categorization (e.g. IEC62304 or IVDR) to be used for Software safety class determination
2. Describe the software development process, and how it is implemented within each project with respect to a specific safety class. For class C of IEC62304, this would include:
    a. Describe the software development planning process and determine the required content of documentation, such as the software development plan.
    b. Describe the software requirement analysis process, for instance, if specific requirement analysis tools are to be used and what information should be included in the software requirement specification of a project.
    c. Describe the software architecture and design process. This could include:
        i. How to determine SOUP and which information to be stored about it.
        ii. How to describe the software architecture and in what detail.
        iii. How to describe the software design and in what detail, e.g. use a specific structure or UML.
        iv. What information is stored in the software architecture document.
    d. Describe the software testing and verification process. This could include:
        i. How to determine which type of tests will be performed
        ii. How these tests and their results will be documented in the software testing plan
3. Describe the software release process which determines the requirements that have to be met before software release, and what information has to be documented within the software release documentation.
4. Describe the configuration management process. This should determine what information needs to be documented in the software configuration and change documentation to ensure traceability within the project.
5. Describe the software problem-solving process which should determine on which level internal and external problems are handled. Moreover, it should be defined, what information is collected and how corresponding changes are documented and communicated within the project.

This check-list provides thus a low-barrier entrance point for academic researchers to follow a tailored software lifecycle process. Note that a Software Life Cycle process should always be embedded into a quality management system, which determines organisational factors, such as responsibilities and roles as well as which documents are required and where these are stored.

5 Conclusion

In this deliverable, we analysed the difficulties of technology transfer from research to commercial manufacturing, focusing on software that is intended to be used in a medical context. We proposed establishing a tailored software life cycle for research organizations, which has the potential to greatly facilitate and speed up a technology transfer in a controlled and predictable way. Being aware that a full software life cycle is not feasible for most research organizations, we proposed a subset of elements of an SLC, which we are convinced will provide a significant benefit while keeping the effort in a range that most organizations can easily handle. Our proposal is centered on procedures for software development planning, software requirement analysis, architectural software design, software unit implementation, integration, and testing, as well as verification and configuration management and problem-solving processes. Depending on the specific needs, the set of elements of an SLC that work best for an organization may differ from what we propose. The fact, however, that an SLC is set up at all and that the elements are deliberately chosen is probably a key factor for

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 826078.

facilitating technology transfer. Our proposal provides a starting point for this, lowering the hurdle for many research organizations to set up some software life cycle.

6 References

1. IEC. IEC 62304:2006+A1:2015 Medical device software - Software life-cycle processes. 2015.
2. European Parliament, Council of the European Union. Regulation (EU) 2017/745 of the European Parliament and the council of 5 April 2017 on medical devices. Official Journal of the European Union. 2017.
3. Medical Device Coordination Group. MDCG 2019-11 Guidance on Qualification and Classification of Software in Regulation (EU) 2017/745 - MDR and Regulation (EU) 2017/746 - IVDR. 2019.
4. Oen DR. Software als Medizinprodukt. Medizin Produkte Recht. 2009;2:55–7.
5. IMDRF SaMD Working Group. IMDRF/ SaMD WG/ N10FINAL2013 Software as a Medical Device. Key Definitions. International Medical Device Regulators Forumdev; 2013.
6. Bundesministerium der Justiz und für Verbraucherschutz. Medizinproduktegestz - MPG § 3 Begriffsbestimmungen. accessed: 11.03.2021 [Internet]. Available from: http://www.gesetze-im-internet.de/mpg/__3.html; Available from: http://www.gesetze-im-internet.de/mpg/__3.html
7. Johner PDC. Software als Medizinprodukt - Software as Medical Device. accessed: 11.03.2021 [Internet]. 2020 Available from: https://www.johner-institut.de/blog/regulatory-affairs/software-als-medizinprodukt-definition/; Available from: https://www.johner-institut.de/blog/regulatory-affairs/software-als-medizinprodukt-definition/
8. European Council. Commission communication in the framework of the implementation of the Council Directive 93 / 42 / EEC of 14 June 1993 concerning medical devices (Publication of titles and references of harmonized standards under the directive). Official Journal of the European Union. 2010.
9. FDA. Recognized Consensus Standards. accessed: 24.03.2021 [Internet]. 2019 Available from: https://www.accessdata.fda.gov/scripts/cdrh/cfdocs/cfstandards/detail.cfm?standard__identification_no=38829; Available from: https://www.accessdata.fda.gov/scripts/cdrh/cfdocs/cfstandards/detail.cfm?standard__identification_no=38829
10. IEC. IEC 82304-1:2016 International Standard Health software. 2016.
11. Johner PDC. Sicherheitsklassen gemäß IEC 62304. accessed: 11.03.2021 [Internet]. 2017 Available from: https://www.johner-institut.de/blog/iec-62304-medizinische-software/sicherheitsklassen-iec-62304; Available from: https://www.johner-institut.de/blog/iec-62304-medizinische-software/sicherheitsklassen-iec-62304
12. Manghani K. Quality assurance: Importance of systems and standard operating procedures. Perspectives in Clinical Research. 2011;2(1):34–7.
13. Johner PDC. Entwicklungsplan versus Entwicklungsprozessbeschreibung. accessed: 15.03.2021 [Internet]. 2015 Available from: https://www.johner-institut.de/blog/iec-62304-medizinische-software/entwicklungsplan-versus-entwicklungsprozessbeschreibung/; Available from: https://www.johner-institut.de/blog/iec-62304-medizinische-software/entwicklungsplan-versus-entwicklungsprozessbeschreibung/
14. Guo H, Zhang R, Chen X, Zou Z, Qu T, Huang G, et al. Quality control in production process of product-service system: A method based on turtle diagram and evaluation model. Procedia CIRP. 2019;83:389–93.
15. Hugh M, Abder-Rahman A, McCaffery F. The Significance of Requirements in Medical Device Software Development. EuroSPI. 2013.
16. Memon M, Jalbani AA, Das Menghwar G, Depar MH, Pathan KT. I2A: AN INTEROPERABILITY & INTEGRATION ARCHITECTURE FOR MEDICAL DEVICE SOFTWARE AND eHEALTH SYSTEMS. Science International. 2016;28(4):3783–7.
17. Mc Hugh M, Cawley O, McCaffcry F, Richardson I, Wang X. An agile V-model for medical device software development to overcome the challenges with plan-driven software development life-

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 826078.

cycles. In: 2013 5th International Workshop on Software Engineering in Health Care, SEHC. IEEE; 2013. p. 12–9.
18. Cao L, Ramesh B. Agile Requirements Engineering Practices: An Empirical Study. In: IEEE Software. IEEE; 2008. p. 60–7.
19. Highsmith J, Cockburn A. Agile Software Development: The Business of Innovation. Computer. 2001;34(9):120–7.
20. Bless M. Scrum und die IEC 62304 Medizinische Software mit agilen Methoden normkonform entwickeln. Vers. 1.2. Baden-Baden; 2013.
21. Büchner F. Wie verifiziert man eine Software-Unit nach IEC 62304 ? Meditronic Journal. 2019;(3):58–61.
22. IEEE Std 610.12-1990. IEEE Standard Glossary of Software Engineering Terminology. IEEE Computer Society; 1990.
23. Bahill AT, Madni AM. Discovering System Requirements. In: Tradeoff Decisions in System Design. Cham: Springer; 2017. p. 373–457.
24. Ebert C. Systematisches Requirements Engineering. Anforderungen ermitteln, dokumentieren, analysieren und verwalten. dpunkt.verlag; 2019.
25. Johner Institut. Schlagwort: Software-Anforderungen IEC 62304 konform dokumentieren. accessed: 16.03.2021 [Internet]. 2019 Available from: https://www.johner-institut.de/blog/tag/software-anforderungen; Available from: https://www.johner-institut.de/blog/tag/software-anforderungen
26. Schachinger P, Johannesson HL. Computer modelling of design specifications. Journal of Engineering Design. 2000;11(4):317–29.
27. Bergsmann J. Requirements Engineering für die agile Softwareentwicklung. 2nd Editio. Heidelberg: dpunkt.verlag; 2018.
28. Rupp C, Simon M, Hocker F. Requirements Engineering und Management. HMD Praxis der Wirtschaftsinformatik. 2009;46:94–103.
29. Hruschka P. Anforderungen ermitteln. In: Business Analysis und Requirements Engineering Produkte und Prozesse nachhaltig verbessern. 2. Auflage. Carl Hanser Verlag; 2019. p. 259–83.
30. Johner Institut. Schlagwort: Interoperabilität: Zusammenarbeiten der IT-Systeme sicherstellen. accessed: 17.03.2021 [Internet]. 2019 Available from: https://www.johner-institut.de/blog/tag/interoperabilitat/; Available from: https://www.johner-institut.de/blog/tag/interoperabilitat/
31. ISO, IEC. ISO/IEC 25010:2011 - Systems and software engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) - System and software quality models. The British Standards Institution; 2011.
32. Bourque P, Fairley RE, editors. Guide to the Software Engineering Body of Knowledge. Swebok. Version 3. IEEE Computer Society; 2014.
33. Richards M, Ford N. Fundamentals of Software Architecture. An Engineering Approach. Sebastopol: O'Reilly Media, Inc.; 2020.
34. Mosleh M, Dalili K, Heydari B. Distributed or Monolithic? A Computational Architecture Decision Framework. In: IEEE Systems Journal. IEEE; 2016. p. 125–36.
35. Jiang ZM, Hassan AE, Holt RC. Visualizing clone cohesion and coupling. In: 2006 13th Asia Pacific Software Engineering Conference (APSEC'06). IEEE; 2006. p. 467–74.
36. Johner Institut. Schlagwort: Softwarekomponenten. accessed: 18.03.2021 [Internet]. 2019 Available from: https://www.johner-institut.de/blog/tag/software-komponente; Available from: https://www.johner-institut.de/blog/tag/software-komponente
37. Hofmeister C, Nord RL, Soni D. Describing software architecture with UML. In: Donohoe P, editor. Software Architecture WICSA 1999 IFIP - The International Federation for Information Processing. Vol 12. Boston, MA: Springer; 1999. p. 145–59.
38. Galitz WO. The Essential Guide to User Interface Design. An Introduction to GUI Design Principles and Techniques. 3rd Editio. Indianapolis: John Wiley & Sons, Inc.; 2007.
39. Johner PDC. Unit Testing und IEC 62304. accessed: 18.03.2021 [Internet]. 2016 Available from: https://www.johner-institut.de/blog/iec-62304-medizinische-software/unit-testing-iec-62304/;

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 826078.

Available from: https://www.johner-institut.de/blog/iec-62304-medizinische-software/unit-testing-iec-62304/
40. Murnane T, Reed K. On the Effectiveness of Mutation Analysis as a Black Box Testing Technique. In: Proceedings of the Australian Software Engineering Conference (ASWEC'01). IEEE; 2001. p. 12–20.
41. Nidhra S, Dondeti J. Black Box and White Box Testing Techniques - A Literature Review. International Journal of Embedded systems and Applications (IJESA). 2012;2(2):29–50.
42. Khan ME. Different Approaches to White Box Testing Technique for Finding Errors. International Journal of Software Engineering and its Applications. 2011;5(3):1–14.
43. Myers GJ, Badgett T, Sandler C, editors. Program Inspections, Walkthroughs, and Reviews. In: The Art Of Software Testing. Third Edit. Hoboken, New Jersey: John Wiley & Sons, Inc; 2012. p. 19–39.
44. Myers GJ, Sandler C, Badgett T, editors. Usability (User) Testing. In: The Art Of Software Testing. Third Edit. Hoboken, New Jersey: John Wiley & Sons, Inc; 2012. p. 143–55.
45. Ahamed SSR. Studying the Feasibility and Importance of Software Testing: An Analysis. International Journal of Engineering Science and Technology. 2009;1(3):119–28.
46. Smilowitz ED, Darnell MJ, Benson AE. Are we overlooking some usability testing methods? A comparison of lab, beta, and forum tests. Behaviour and Information Technology. 1994;13(1–2):183–90.
47. French-Mowat E, Burnett J. How are medical devices regulated in the European Union? Journal of the Royal Society of Medicine. 2012;105:22–8.
48. Johner PDC. Software-Freigabe: ein „beliebtes“ Missverständnis. accessed: 18.03.2021 [Internet]. 2017 Available from: https://www.johner-institut.de/blog/iec-62304-medizinische-software/software-freigabe/; Available from: https://www.johner-institut.de/blog/iec-62304-medizinische-software/software-freigabe/
49. Riemenschneider M, Wienbeck J, Scherag A, Heider D. Data Science for Molecular Diagnostics Applications: From Academia to Clinic to Industry. Systems Medicine. 2018;1(1):13–7.
50. Gotel O, Cleland-Huang J, Hayes JH, Zisman A, Egyed A, Grünbacher P, et al. Traceability fundamentals. In: Cleland-Huang J., Gotel O. ZA, editor. Software and Systems Traceability. London: Springer; 2012.
51. Regan G, McCaffery F, Mc Daid K, Flood D. Medical device standards' requirements for traceability during the software development life-cycle and implementation of a traceability assessment model. Computer Standards and Interfaces. 2013;36(1):3–9.
52. Gotel OCZ, Finkelstein ACW. Analysis of the requirements traceability problem. In: Proceedings of the IEEE International Conference on Requirements Engineering. IEEE; 1994. p. 94–101.
53. Johner PDC. Stakeholder Anforderungen / Requirements ~ Definion & Verwendung. accessed: 31.05.2021 [Internet]. 2015 Available from: , https://www.johner-institut.de/blog/tag/requirements/
54. Johner PDC. Design Input und Output ~ Definition und Requirements. accessed: 31.05.2021 [Internet]. 2015 Available from: https://www.johner-institut.de/blog/fda/design-input/,2017
55. Medsoto GmbH. MeddPack. accessed: 30.05.2021 Available from: http://www.medsoto.de/produkte-leistungen/medpack?lang=en

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 826078.

7 Table of acronyms and definitions

AAMI	Association for the Advancement of Medical Instrumentation
concentris	concentris research management GmbH
GND	Gnome Design SRL
GUI	Graphical User Interface
IEC	International Electrotechnical Commission
IEEE	Institute of Electrical and Electronics Engineers
IMDRF	International Medical Device Regulators Forum
IVDR	In Vitro Diagnostic Medical Devices Regulation
LGS	LeGacy Software
LS	Life Cycle
MDCG	Medical Device Coordination Group
MDR	Medical Device Regulation
MDSW	Medical Device Software
MS	Milestone
MUG	Medizinische Universitaet Graz
Patients	In this deliverable, we use the term "patients" for all research subjects. In FeatureCloud, we will focus on patients, as this is already the most vulnerable case scenario and this is where most primary data is available to us. Admittedly, some research subjects participate in clinical trials but not as patients but as healthy individuals, usually on a voluntary basis and are therefore not dependent on the physicians who care for them. Thus to increase readability, we simply refer to them as "patients".
QMS	Quality Management System
RI	Research Institute AG & Co. KG
SaMD	Software as a Medical Device
SBA	SBA Research Gemeinnutzige GmbH
SDU	Syddansk Universitet
SLC	Software Life Cycle
SOP	Standard Operating Procedure
SOUP	Software of Unknown Provenance
TUM	Technische Universitaet Muenchen
UM	Universiteit Maastricht
UML	Unified Modeling Language
UMR	Philipps Universitaet Marburg
WP	Work package

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 826078.