

Chapter 3 / Capítulo 3

Primum non nocere: Software engineering
for medicine
ISBN: 978-9915-9876-8-2
DOI: 10.62486/978-9915-9876-8-2.Ch03
Pages: 50-62



© 2026 The authors. This is an open access
book chapter distributed under the terms of
the Creative Commons Attribution (CC BY)
4.0 License.

Chapter 3. IEC 62304: Medical software lifecycle

Annier Jesús Fajardo Quesada  

Eduardo Antonio Hernández González 

René Herrero Pacheco 

Rolando Javier Álvarez Pérez 

1 Granma University of Medical Sciences. Granma, Cuba.

2 University of Medical Sciences of Pinar del Río. Pinar del Río, Cuba.

“Classifying medical software by risk level is not a bureaucratic exercise, but an act of honesty: recognizing that not all failures have the same consequences and that, therefore, not all developments deserve the same level of scrutiny.”

Imagine we are in a planning meeting for a new healthcare software project. Someone asks, “What level of regulatory requirements should we apply?” Eyes dart around; no one is sure of the answer. Some believe that since it is just software, it should be a light-touch approach. Others, more cautious, suggest applying everything the standards require, just in case. The discussion drags on aimlessly because a fundamental element is missing: the actual risk of the product has not been analyzed. The most basic question has not been answered: what happens if this fails? Depending on the answer, regulatory requirements can vary enormously, from a relatively simple process to a development process with exhaustive controls at every stage. This gradation is not an invention of each team but is clearly defined in an international standard that forms the backbone of medical software development: IEC 62304.

IEC 62304 is likely the most important standard that any medical software developer must know.⁽¹⁾ Its full title is “Medical device software—Software life cycle processes,” and it establishes a framework for all development activities, from initial planning through maintenance once the product is on the market. What makes this standard particularly valuable is that it is not an abstract set of good intentions but a concrete specification of tasks, documentation, and responsibilities that must be fulfilled depending on the software’s risk level. It is, in a sense, a practical guide that says: if your software is low-risk, do this; if it is medium-risk, do that; if it is high-risk, go this far. This proportionality is its greatest strength and makes it applicable to a wide range of products, from the simplest to the most critical.

In this chapter, we explore in detail the fundamental aspects of IEC 62304, starting with the concept that underpins everything: software safety classification. We will see what each class means, how to determine which one applies to each product, and what practical implications that classification has for the development process. It is not a matter of memorizing a table but of understanding the logic underlying the classification because that logic is the same one that should guide any design decision when patient safety is at stake. Understanding IEC 62304 means understanding how to build medical software in a systematic and responsible manner, and it is also the first step toward navigating with confidence the regulatory maze that so many developers fear.

3.1 SAFETY CLASSIFICATION: CLASSES A, B, AND C

To understand the safety classification in IEC 62304, it is helpful to start with a simple yet uncomfortable question: what happens if this software fails? Not if it fails in a specific way, not if it fails under specific circumstances, but in general, if the software stops doing what it is supposed to do, or does something it is not supposed to do, what are the consequences for the patient, for the user, for the environment? The answer to this question is the fundamental factor that determines the software’s safety class. Because a failure in an app that helps remind users to take their medication—which might lead to a missed dose—is not the same as a failure in a drug infusion system, which could cause a lethal overdose. The standard recognizes this difference and establishes three levels of increasing requirements on the basis of the potential severity of the consequences.

Class A is the lowest and corresponds to software whose failure cannot cause injury or harm to health. This may sound strange: if it cannot cause harm, why are we talking about medical software? The answer is that Class A software may be part of a medical device, but its contribution to safety is indirect or noncritical. For example, an application that keeps track of patient appointments or a program that manages a hospital’s inventory can be considered Class A if its failure does not directly affect people’s health. Even within a more complex device, there may be software components that are Class A because their malfunction has no clinical consequences. The key is that, for this class, patient safety does not depend on the software functioning correctly.

3.1.1 THE RISK PYRAMID: FROM HARMLESSNESS TO MORTALITY

Class B is the intermediate level and corresponds to software whose failure may cause injury but not serious injury or death. The definition of “injury” in this context is broad and includes both physical and psychological harm, provided that it is reversible or minor in nature. For example,

software that calculates nonlethal drug doses or a vital sign monitoring system that may trigger false alarms could be classified as Class B if an error causes discomfort or unnecessary treatments but does not endanger the patient's life. The distinction between Class B and Class C is not always clear-cut and requires a careful analysis of the potential consequences of failures, taking into account not only the direct effect but also the chains of events that may be triggered.

Class C is the highest category and applies to software whose failure can cause death or irreversible serious injury. This category includes the most critical systems: software that controls life-support equipment, algorithms that guide cancer treatments, high-risk drug infusion systems, and diagnostic applications that can lead to erroneous therapeutic decisions. For these products, the standards are the highest because people's lives are at stake. Every line of code, every design decision, and every test must be executed with the utmost rigor, and the documentation must be exhaustive to allow for audits and complete traceability. It is not that Class C developers are more careful but rather that the process within which they work is designed to minimize the probability of errors to the utmost extent.

The assignment of the safety class is not something the manufacturer determines arbitrarily but must be based on a risk analysis conducted in accordance with ISO 14971, which we explore in the next chapter. This analysis identifies the potential hazards associated with the use of the software, estimates the probability of their occurrence and the severity of their consequences, and determines whether existing control measures are sufficient. The safety class is derived from the severity of the consequences of failures, considering the worst reasonably foreseeable scenario. It is not a matter of being optimistic but of being realistic: if there is a possibility that a failure could cause death, the software is Class C, even if that possibility is remote. The standard prefers to err on the side of caution rather than underestimate the risks.

3.1.2 MODULAR CLASSIFICATION AND THE CONCEPT OF INDEPENDENCE

An important aspect is that the classification applies to the software as a whole, but it can also be applied to individual components within a larger system. A medical device may contain several software elements with different safety classes. For example, a monitoring system in an intensive care unit may have a vital sign display module that is Class B (because poor display can lead to noncritical erroneous decisions) and an alarm control module that is Class C (because an alarm that does not sound when it should have lethal consequences). In these cases, the development must meet the requirements of the highest class for the system as a whole, unless a genuine independence between the modules can be demonstrated that justifies differentiated treatment.

Independence between modules is a technical concept with significant regulatory implications. Two software modules are considered independent if a failure in one cannot cause a failure in the other and if the operation of each can be verified separately. In practice, this means that they must run in separate memory spaces, communicate through well-defined interfaces, and not share critical resources. If this independence can be demonstrated, then it is possible to develop a Class C module with all necessary controls and another Class B module with fewer requirements, even within the same device. However, the burden of proof lies with the manufacturer, who must demonstrate that the independence is genuine and that there are no hidden pathways through which a failure in the lower-risk module could affect the higher-risk module.

Demonstrating independence requires rigorous architectural analysis and specific testing. How the modules are isolated in terms of memory, processor time, and communications must be documented. Fault injection tests must be performed to verify that an error in the lower-class module does not propagate to the higher-class module. This independence must be maintained throughout the entire lifecycle, with reviews conducted whenever changes occur that could affect the architecture. The temptation to "optimize" by sharing resources between modules must be resisted when it compromises the independence that justifies the differentiated classification.

3.1.3 PRACTICAL IMPLICATIONS: FROM ANALYSIS TO DEVELOPMENT

Determining the safety class has enormous practical implications for the development process. IEC 62304 establishes a set of mandatory activities for each class such that the higher the risk is, the more activities must be performed and the greater the level of detail needed.

For example, for Class A software, the documentation requirements are minimal; for Class B, traceability between requirements and tests is needed; for Class C, more comprehensive requirements for detailed architectural design, code review, and test coverage analysis are added. This gradation ensures that low-risk products are not bogged down by unnecessary bureaucracy, while guaranteeing that high-risk products receive the attention they deserve.

For development teams, understanding the security class from the outset is critical because it shapes many subsequent decisions. It is not the same to design the architecture knowing that you will have to demonstrate independence between modules as it is to do so without that concern. It is not the same to plan testing, knowing that code coverage will be needed, as it is to do so with more relaxed requirements. It is not the same to organize documentation knowing it will be thoroughly audited, as it is to do so with the confidence that it will be only briefly reviewed. That is why the security classification is not something to be left until the end but must be determined in the early stages of the project and used as a guide for all subsequent decisions.

A common mistake among novice developers is to believe that the safety class is something negotiated with the regulator to obtain more lenient requirements. This view is profoundly mistaken and potentially dangerous. The safety class is not a variable in the decision-making process but a consequence of the risk analysis. If the analysis shows that a failure could cause death, the software is Class C, regardless of what the manufacturer prefers or negotiates with the regulator. Attempting to classify software that should be Class C as Class B is, at best, a lack of understanding of the standard and, at worst, an unrelated irresponsibility that can have tragic consequences. Honesty in classification is the first ethical requirement of medical software development.

3.1.4 CHANGES, UPDATES, AND CONTINUOUS MONITORING

The relationship between safety classification and software updates deserves special mention. When existing software is modified, its classification must be reevaluated because changes can introduce new risks or alter existing ones. An update that adds a critical functionality may increase the safety class; one that removes dangerous functionalities may lower it. This reassessment must be documented, and if the class changes, a new certification or, at a minimum, a review by the regulator may be needed. That is why it is important to keep the risk analysis up to date throughout the product's entire lifecycle, not just as an initial formality but as an ongoing practice.

For physicians involved in development projects, understanding the safety classification is also relevant because their clinical knowledge is essential for correctly assessing the consequences of potential failures. An engineer may not know what it means for a patient to experience a certain complication or how serious it actually is. The physician provides that perspective, helping to ensure that the risk analysis is realistic and comprehensive. That is why it is important for clinicians to be involved in the early stages of the project, when the safety class is being determined, and for their voice to be heard and respected. Classification is not a technical decision but a clinical-technical decision, and it requires collaboration between both worlds.

One aspect that often raises questions is the classification of software that has no direct medical use but can indirectly affect safety. For example, an appointment management system that, if it fails, could cause a patient to miss a critical appointment. Is that Class B or Class C? The answer depends on the severity of the consequences of that missed appointment. If it is for routine follow-up of a stable chronic condition, it is likely Class B. If it is for administering a treatment with a narrow therapeutic window, it could be Class C. There are no automatic answers; each case must be analyzed in its context, considering the worst reasonably foreseeable scenario. This ambiguity is uncomfortable for those seeking clear rules, but it is inherent in a field where clinical reality is complex and variable.

Experience shows that many teams tend to underclassify their software, whether due to ignorance, optimism, or commercial pressure to reduce costs. This tendency is dangerous because it leads to development processes that are insufficient for the actual risk, increasing the likelihood of incidents occurring. That is why regulators pay special attention to the justification for the classification and require that it be well documented and supported by rigorous analysis. A manufacturer that cannot explain why its software is Class B and not Class C will arouse suspicion and will likely be subject to more detailed scrutiny. Transparency and rigor in classification are also ways to protect against future claims.

It is worth remembering that security classification is not an end in itself but a means to a

higher end: aligning the development effort with the product's actual risk. It is not about creating obstacles but about focusing resources where they are most needed. Class C software deserves all the attention, all the controls, and all the documentation because human lives depend on it. Class A medical software can be developed using lighter processes because its failures will not cause harm. This distinction, far from being discriminatory, is deeply ethical: it recognizes that resources are limited and must be invested where they can best protect patients. Understanding and applying it correctly is the first step toward building responsible medical software and toward avoiding wasting effort on unnecessary bureaucracy while ignoring real risks.

If the safety classification determines how much control we need, but the risk analysis underpinning it depends on a subjective assessment of clinical consequences, how do we ensure that this assessment is consistent across different teams and regulators, avoiding both dangerous underclassification and paralyzing overclassification?

3.2 PRACTICAL IMPLICATIONS OF EACH CLASS

"The safety class is not a punishment, but a protection: for the patient, for the physician, and for the developer themselves."

Determining whether software is Class A, B, or C is not an academic exercise that ends with a report that is filed away and forgotten. In contrast, it is a decision that permeates every aspect of development, from how requirements are documented to the depth of testing that must be performed, including system architecture, change management, and even team composition. Knowing the safety class is similar to having a map that shows which route to take: the paths for Class A are simple trails with few demands; those for Class B require slightly more preparation and equipment; and those for Class C are multilane highways with constant signage and exhaustive checks. Ignoring this map and treating all developments the same inevitably leads to one of two problems: either excessively burdensome processes are applied to a low-risk product, making it economically unviable, or insufficient processes are applied to a high-risk product, putting patients at risk.

3.2.1 PLANNING AND DOCUMENTATION: THE ESCALATION OF REQUIREMENTS

To understand the practical implications of each class, we must examine the main activities of the development cycle and see how they change depending on the risk level. Let us start with **planning**. For any class, IEC 62304 requires a software development plan, but the level of detail required varies. For Class A, it can be a simple document that broadly outlines the phases, deliverables, and responsible parties. For Class B, the plan must be more detailed, including timelines, milestones, and a description of the verification and validation processes. For Class C, the plan must be comprehensive, with unique identifiers for each activity, a precise definition of the input and output criteria for each phase, and a detailed description of how traceability will be ensured among all development elements. Additionally, for Class C, the plan must be formally reviewed and approved by management before work begins.

Requirement management is another area where the differences between classes are notable. For Class A, a list of functional requirements is sufficient, without the need for extensive formality. For Class B, requirements must be documented in a structured manner, with unique identifiers, and traceability must be established between them and the tests that verify them. For Class C, in addition to the above, requirements must be analyzed from a risk perspective, identifying those that are critical to safety, and the justification for each requirement must be documented (), explaining why it is necessary and what alternatives were considered. Additionally, for Class C, changes to requirements must undergo a formal change control process, including impact assessment and approval prior to implementation.

Architectural design is perhaps where the differences between classes become most apparent. For Class A, design documentation is rarely needed; it is sufficient for the code to be understandable. For Class B, a high-level description of the architecture is needed, identifying the main components and their interactions. For Class C, the design must be detailed and comprehensive, including component diagrams, interface descriptions, specifications of critical algorithms, and an analysis of how the architecture contributes to security. Additionally, for Class C, the design must undergo formal peer review, and design decisions must be justified and documented. The reason for this requirement is that in high-risk software, design errors are

particularly dangerous because they affect the entire system and are difficult to correct once development is well underway.

3.2.2 IMPLEMENTATION AND TESTING: CODE TO THOROUGH VERIFICATION

Implementation—that is, writing the code—also has different levels of requirements depending on the class. For Class A, there are no specific requirements beyond those imposed by the team's professional standards. For Class B, it is recommended to follow coding standards that improve the readability and maintainability of the code. For Class C, coding standards are mandatory and must also include specific guidelines to prevent common errors in critical software, such as memory management, error handling, or overflow prevention. Additionally, for Class C, the code must be systematically peerreviewed, and these reviews must be documented. In some cases, it may be necessary to use static code analysis tools to automatically detect potential defects.

Integration and testing are likely the areas where the differences between classes are most quantifiable. For Class A, functional tests that demonstrate that the software does what is expected are sufficient. For Class B, in addition to functional tests, integration tests are needed to verify that components work correctly together, and regression tests are needed to ensure that changes have not introduced new errors. For Class C, the level of rigor increases significantly: unit tests with code coverage (typically at least 80% of statements) are needed, along with comprehensive integration tests covering all interfaces, system tests simulating real-world usage conditions, and robustness tests verifying the software's behavior under abnormal conditions. Furthermore, all these tests must be documented, with test cases traceable to requirements, and the results must be retained for potential auditing.

Risk management, which we explore in detail in the next chapter, also has different implications depending on the class. For Class A, the risk analysis can be brief, focusing on identifying obvious hazards. For Class B, a systematic, documented analysis is needed that identifies hazards, estimates their probability and severity, and defines control measures. For Class C, the risk analysis must be comprehensive and ongoing and updated throughout the entire lifecycle, and control measures must be verified and validated with particular rigor. Furthermore, for Class C, residual risks—those that remain after applying control measures—must be documented and explicitly communicated to users.

3.2.3 DOCUMENTATION, CHANGES, AND KNOWLEDGE MANAGEMENT

Documentation, that aspect of development that elicits so many complaints among programmers, is another area where the differences between classes are notable. For Class A, documentation may be minimal: a user manual, a design summary, and basic test records. For Class B, the documentation must include the development plan, requirements, high-level design, test cases, results, and a change log. For Class C, all of the above is supplemented by detailed design documentation, code reviews, a comprehensive risk analysis, complete traceability between all the elements, and a design history file (DHF) that allows the entire development process to be reconstructed. Furthermore, for Class C, the documentation must be structured in a way that is easily navigable and understandable to external auditors.

Change management is another area where the differences are marked. For Class A, changes can be managed informally, with a basic record of what has been modified. For Class B, a documented change control process is needed, including impact assessment, verification that the change has been implemented correctly, and updating of the affected documentation. For Class C, the change process must be particularly rigorous, involving a detailed assessment of the impact on security, formal review by a change control board, thorough verification through regression testing, and validation that the change has not introduced new risks. Additionally, for Class C, significant changes may require new regulatory approval prior to implementation.

Team composition and required competencies also vary by class. For Class A, there are no specific requirements regarding staff training. For Class B, it is recommended that those involved have training and experience in software development and that there be a clear assignment of responsibilities. For Class C, personnel are required to have demonstrated competencies in critical software development; there must be a clear separation of duties (for example, between developers and testers), and records of each person's training and experience must be maintained. In many

Class C projects, it is common to have a quality manager dedicated exclusively to overseeing process compliance.

3.2.4 INTERACTION WITH REGULATORS AND MANAGEMENT OF MIXED PROJECTS

The **relationship with the regulator** also varies by class. For Class A products, interaction with the regulator is generally not required beyond product registration. For Class B, it may be necessary to submit technical documentation demonstrating compliance with essential requirements. For Class C, interaction with the regulator is typically intensive, including the submission of complete dossiers, the response to detailed questions, and, in many cases, onsite audits of the development process. Manufacturers of Class C products must be prepared to dedicate significant resources to this interaction and to maintain fluid and transparent communication with the authorities.

For teams developing products of different classes, a common challenge is managing the coexistence of varying levels of regulatory requirements. It may happen, for example, that a Class C product uses a third-party library that is Class B or that a system integrates modules of different classes. In these cases, the standard allows for some flexibility, provided that the lower-class modules do not affect the safety of the higher-class modules. However, the burden of proof lies with the manufacturer, who must document how this interaction has been analyzed and what measures have been taken to ensure that a failure in a lower-class module does not compromise the safety of the whole system. This often requires additional analysis and, in some cases, specific testing.

One aspect that is often underestimated is the impact of the safety class on project timelines and costs. The development of Class C software can easily cost several times more than the development of Class A software of similar functional complexity because of all the additional processes, exhaustive documentation, and extra testing. This is not a flaw in the standard but an inevitable consequence of the higher safety requirements. Manufacturers must be realistic about these costs from the outset and not commit to timelines or budgets that would only be feasible for a lower-risk product. Underestimating the required effort is among the main causes of failure in medical software projects and is often due to a misunderstanding of the practical implications of the safety class.

Understanding the implications of each class helps clinicians grasp why the process is so slow or so costly. When an engineer explains that a certain feature requires more time because the software is Class C, they are not making excuses but describing regulatory reality. A clinician who understands this can adjust their expectations and help prioritize what is truly important, freeing up resources for what genuinely impacts safety. They can also help identify which features are truly critical and which could be implemented more simply, contributing to more efficient development without compromising safety.

Industry experience shows that many teams, especially those new to medical software, tend to underestimate the implications of the safety class. They believe that they can develop a Class C product using the same methods they use for web applications, and they encounter unpleasant surprises in regard to the time for certification. The documentation is insufficient, the tests lack the required coverage, and traceability is nonexistent. The result is costly delays and, in the worst-case scenario, the abandonment of the project. That is why it is so important to internalize from the outset that the safety class is not a label but a work program that determines how the product will be developed in each of its phases.

The practical implications of each class are not an end in themselves but a means to ensure that the software is secure and effective. The additional activities required for Class C are not empty bureaucracy but responses to real risks identified over decades of experience. When a developer complains about having to document a code review or having to run code coverage tests, it is worth reminding them that these practices exist because someone, at some point, made a mistake that could have been avoided with them. The safety class is not a punishment but a safeguard: for the patient, who receives a more reliable product; for the doctor, who can trust the tool they use; and for the developer themselves, who can sleep soundly knowing they did everything possible to prevent harm.

3.3 MANDATORY DOCUMENTATION VS. USEFUL DOCUMENTATION

There is a scene that plays out in different ways in every medical software project involving

inexperienced teams. At some point, someone utters the phrase “We have to document this because the standard requires it.” Immediately afterward, a developer, with a distant look in their eyes and a sigh of resignation, begins filling out templates, writing paragraphs they know no one will read, generating documents that clutter digital folders without adding any real value to the product. This scene is the clearest manifestation of something that plagues many projects: the confusion between mandatory documentation and useful documentation. Documentation is created because it has to be done, not because it serves a purpose. When this happens, documentation becomes a burden, a hindrance, and a source of frustration that distracts developers from what truly matters: building software that works and is safe.

To understand the difference between mandatory documentation and useful documentation, it is worth returning to the following fundamental question: What is the purpose of documentation in medical software development? The answer has several layers that deserve careful exploration. It helps the team develop a shared understanding of what is being built, creating a common language that reduces misunderstandings and aligns efforts. It helps ensure that those who come after can understand the decisions that were made, preventing knowledge from being lost when original team members move on to other projects. It serves to demonstrate to regulators that the process was carried out correctly, providing tangible evidence that best practices were followed. It helps investigate incidents when something goes wrong, allowing the chain of events leading to the problem to be reconstructed. It helps manage changes without introducing errors, providing a system map that guides modifications. Ultimately, it helps make the software safer and more reliable.

Documentation that fulfills any of these functions is useful. Documentation that fulfills none of them, no matter how strictly required by the standard, is an empty bureaucracy. The art of managing documentation in regulated environments consists precisely of maximizing the former and minimizing the latter.

3.3.1 THE FUNDAMENTALS OF PURPOSEFUL DOCUMENTATION

IEC 62304 establishes documentation requirements for each class of software but does not specify the format, length, or level of detail.⁽¹⁾ This apparent vagueness is actually profound wisdom because the standard recognizes that documentation must be tailored to the product, the team, and the specific context of each project. What might require hundreds of pages for a complex Class C product may be just a dozen for a simple Class A product. What matters is not quantity but quality: that the documentation contains the information necessary to fulfill its purpose and that it is organized in such a way that it can be found and understood when needed. A common—and costly—mistake is to believe that “more documentation” is synonymous with “better compliance.” This is not the case. Excessive documentation can be just as harmful as insufficient documentation because it makes it difficult to find what is important, consumes resources that could be devoted to activities that truly add value, and creates a false sense of security that distracts from real risks.

To distinguish between mandatory documentation and useful documentation, it is helpful to ask yourself some practical questions that act as a healthy filter. Who is going to read this? If the answer is “no one,” it is likely an empty bureaucracy that exists solely to meet a formal requirement. What is it going to be used for? If there is no clear purpose, it is likely an empty bureaucracy generated by organizational inertia. Will it be kept up to date? If the answer is “no,” because it is too burdensome to maintain, then it’s empty bureaucracy disguised as compliance, which will also be dangerously misleading when consulted in the future. Does it help make better decisions? If not, it is an empty bureaucracy that adds no value to the development process. Does it facilitate the onboarding of new team members? If not, it is empty, une bureaucracy that perpetuates dependence on tacit knowledge. Does it allow for incident investigation? If not, it is empty bureaucracy that leaves the team unprotected when problems arise. Useful documentation passes this test; documentation that does not, regardless of how mandatory it may seem, should be questioned and, if possible, eliminated or simplified.

A concrete example can help illustrate this distinction in a tangible way. Let us imagine a design requirement for a medical software feature. The mandatory documentation could be an entry in a requirements management tool with a unique identifier, a textual description, and a status indicating whether it is implemented, in progress, or pending. That is useful if it allows us to know which requirements are implemented, which are pending, and which have changed, providing a clear view of the project’s status. However, if each requirement is accompanied by a

multipage “impact analysis,” and that analysis merely repeats in other words, what is already in the requirement, then we are dealing with an empty bureaucracy that consumes time without providing new information. An impact analysis is useful when it truly adds value, when it identifies specific risks, when it evaluates implementation alternatives, or when it estimates necessary resources. If it’s merely a copy-and-paste formality, it’s better not to do it and instead devote that time to more productive activities.

Another example that resonates with many teams: meeting minutes. In many projects, there is a requirement to document all meetings, including attendee lists, agenda items, and decisions made. This can be useful for meetings where important decisions affecting design or planning are made because it helps them recall why certain decisions were made, preventing those decisions from being repeatedly questioned or the arguments supporting them from being forgotten. However, for daily coordination meetings, where the day’s work is simply assigned or operational obstacles are resolved, detailed minutes are empty bureaucracy that adds an administrative burden without corresponding value. A brief note in the task tracking system, updating statuses and assignments, is more than enough. The key is to tailor the level of documentation to the importance of what is being documented, applying the principle of proportionality that should govern all medical software development.

Traceability is another area where the distinction between mandatory and useful is crucial and frequently misunderstood. The standard requires traceability between requirements, design, code, and tests, and this is useful because it allows us to know which tests to run when a requirement is modified or which requirements may be affected when a piece of code is changed, facilitating impact analysis and reducing the risk of undetected side effects. However, traceability can become empty bureaucracy if it is maintained manually, with very large matrices in spreadsheets that no one actually updates, or if it is needed at unnecessary levels of detail that generate only noise. Useful traceability is that which is automatically generated by development tools that can be consulted interactively and that is actually used in daily work to make informed decisions. Traceability that exists only in static documents generated at the end of the project—likely by copying and pasting from other documents—is, at best, an illusion of control that does not truly protect the patient.

3.3.2 REVIEWS, RISKS, AND THE ROLE OF TECHNOLOGY

Code reviews are another interesting example where the tension between mandatory and useful reviews is clearly evident. The standard requires that code be reviewed, especially for Class C (), and this is useful because reviews detect errors that automated tests miss because they foster knowledge sharing among team members and because they spread understanding of the system beyond the original author. However, review documentation can be light and effective: a record of which files were reviewed, who the reviewer was, what issues were found, and how they were resolved. This information is sufficient to demonstrate compliance and to learn from recurring errors. If, instead, a detailed report is required for each review, with a thorough analysis of every line of code, the documentation becomes a burden that discourages frequent reviews—which are precisely the most valuable ones. The balance lies in documenting just enough to provide evidence that the review was conducted and to record the main findings, without drowning the process in a bureaucracy that causes developers to avoid reviews or perform them superficially.

Risk management is particularly prone to empty bureaucracy, perhaps because its abstract nature makes it more difficult to anchor in operational reality. The standard requires a documented risk analysis, and many organizations respond by generating massive risk matrices with hundreds of rows, many of which are repetitive, generic, or irrelevant to the specific product. The matrix is filled out once it is filled away in some repositories, and no one looks at it again until the next audit, when a few dates are updated and little else. This is an empty bureaucracy in its purest form, which meets the formal requirements but does not protect the patient. Useful risk management is the kind that is integrated into daily development; it is updated when a new feature is introduced that could create new hazards that guide architectural design decisions and that is consulted when an unexpected problem arises to see if it was anticipated. A small but living matrix, one that truly reflects the product’s risks and evolves with it, is worth more than a massive but dead tome that only serves to accumulate digital dust.

A useful strategy is to start with a minimal core of truly useful documentation and to add only what proves its value in practice. Ask yourself: What information do we need to coordinate effectively as a team? What would we need if a key team member left tomorrow? What would we

need to investigate an incident affecting a patient? What would we need to convince an auditor that we did our job well? That is the documentation that truly matters—the documentation that addresses real needs. Everything else is incidental, and if it does not add value, it can and should be eliminated or simplified. This minimalist approach not only saves work but also improves the final quality because it focuses attention and resources on what is truly relevant to patient safety.

Technology can go a long way toward reducing the burden of documentation while simultaneously making it more useful. Modern development tools allow for the automatic generation of many of the records required by the standard: traceability between code and tests, test execution results, code coverage metrics, and a history of changes along with their authors and justifications. Leveraging these capabilities is essential to ensure that documentation does not become a manual burden that distracts from the actual development work. A team that properly configures its continuous integration pipeline can automatically generate most of the evidence needed to demonstrate compliance with verification requirements. Manual documentation is then reserved for what truly requires human judgment, contextual analysis, and decision-making: architectural design decisions, risk analyses that evaluate alternative options, and justifications for why something was done in a certain way when other options were possible.

One aspect that is often overlooked, especially by those who view documentation as an external imposition, is that useful documentation is not just for regulators but also for the development team itself. A well-documented design helps new developers speed up more quickly, reducing the time it takes for them to become productive and distribute knowledge more equitably. A well-maintained record of decisions prevents the repetition of discussions settled months ago, saving time and reducing interpersonal conflicts. Clearly, specified requirements reduce misunderstandings among team members, between developers and testers, and between the technical team and clinical stakeholders. Good test documentation facilitates debugging when something goes wrong, allowing us to understand what was being tested and why. From this perspective, documentation is not a burden imposed from outside by bureaucratic entities but a tool for working better, more efficiently, and with fewer errors. Teams that internalize this vision stop complaining about documentation and begin to value it for what it is: an investment in their own effectiveness.

3.3.3 DOCUMENTATION AS A LIVING PRACTICE: PHYSICIANS, AUDITORS, AND EVOLUTION

For the doctors who participate in these projects, understanding what documentation is truly useful helps them focus their contributions effectively and prevent them from feeling that their time is wasted on paperwork. When an engineer asks to help draft a clinical requirement, it is not a meaningless bureaucratic exercise but a crucial opportunity to ensure that the software will do what medical practice truly needs, capturing nuances that only someone who has been at the patient's bedside can know. When a risk analysis is reviewed, it is not an abstract theoretical activity but a way to identify hazards that only deep clinical knowledge can anticipate, such as nonobvious drug interactions or patient behaviors in stressful situations.

Physicians who actively participate in creating useful documentation directly contribute to product safety, not wasting time on worthless paperwork. This understanding transforms medical involvement from an uncomfortable obligation into a valued and valuable contribution.

The relationship with auditors can also be approached more productively if a clear distinction is made between mandatory and useful documentation and if there is an understanding of what auditors are actually looking for. An experienced auditor, who has seen dozens of projects, is not impressed by the quantity of documents but by their quality and consistency. They prefer to see a few well-crafted documents that truly reflect the team's work and demonstrate that they understand what they are doing, rather than a mountain of paperwork that has clearly been generated solely for the audit, with no real connection to development. Transparency and honesty are always the best policies in these interactions.

If a document is missing, it's better to acknowledge it openly and explain why that decision was made than to try to cover it up with fake, incomplete, or clearly copy-pasted documents from other projects. Auditors are not looking for perfect teams that never make mistakes; they are looking for teams that understand what they are doing, are able to explain it coherently, and demonstrate that they have a system for managing quality and security.

The evolution of documentation throughout the project also deserves careful attention because documentation that is useful in one phase may be useless in another. At the beginning, when ideas are vague and the product is not yet clearly defined, documentation can be light,

exploratory, almost schematic, and serve as a thinking tool rather than a definitive record. As the design solidifies and important decisions are made, the documentation must become more precise, detailed, and formal, reflecting the system's growing maturity. With respect to release approaches, the documentation must be complete, reviewed, validated, and ready for use in operation and maintenance. After release, the documentation must be kept up to date with changes, evolving alongside the software it describes. This natural evolution is easier to manage if one understands that documentation is not a static product delivered once but a living reflection of the project's state that requires ongoing care. Tools that allow documentation to be versioned alongside the code, keeping them in sync, greatly facilitate this dynamic management and reduce the risk of inconsistencies.

A common mistake, especially in organizations with a highly formalistic culture, is to believe that useful documentation must necessarily be extensive, densely technical, and written in solemn language. This is not the case. Good documentation can be concise, even schematic, as long as it contains the essential information and is organized logically. A well-designed diagram showing the system architecture and the relationships between components is worth more than ten pages of descriptive text that few will read. A well-structured checklist that guides users through a complex process can replace a comprehensive manual in which no one consults. Well-placed and well-written code comments can eliminate the need for external documentation to explain local implementation decisions. The key is to choose the right format for each type of information, maintain consistency across different formats, and always ask which format will make it easiest to find and understand the information when needed.

Documentation is not an end in itself but a means to a higher end: safe and effective software that improves patient care. When documentation ceases to serve that purpose—when it becomes an obstacle that slows down development without improving quality or a source of frustration that demoralizes the team—something is fundamentally wrong with how it is being approached. We may be documenting things that do not truly matter or documenting inefficiently in a way that does not meet real needs, or we may have lost sight of the ultimate purpose of all this effort. In such cases, the solution is not to abandon documentation and descend into chaos but to critically rethink it. To ask ourselves again, honestly: what exactly is the purpose of this? Who does it actually help? How can we make it better, more useful, and less burdensome? This ongoing reflection, this willingness to question and improve, is what allows documentation to remain useful throughout the entire product lifecycle, rather than becoming that bureaucratic burden that developers so fear and that does so little to protect patients.

Once we have established how to properly document development, how do we integrate risk management so that it is not an isolated procedure but the true driving force behind design decisions?

REFERENCES

IEC 62304:2006+AMD1:2015 CSV | IEC [Internet]. [cited 2026-05-13]. Available from: <https://webstore.iec.ch/en/publication/22794>