

## Chapter 2 / Capítulo 2

Primum non nocere: Software engineering  
for medicine  
ISBN: 978-9915-9876-8-2  
DOI: 10.62486/978-9915-9876-8-2.Ch02  
Pages: 27-49



© 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 2. Introduction to Regulated Environments

**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.

---

---

“Regulation in digital health is not an obstacle that innovators must overcome but rather a map that shows them where the cliffs are. Ignoring it does not make them disappear; it only ensures that the fall will be more painful.”

There is a widespread narrative in the world of technological innovation that portrays regulation as the enemy. In this story, regulators are slow-moving bureaucrats who do not understand technology, who impose absurd obstacles, who hinder progress, and who protect established interests against the brave entrepreneurs who want to change the world. It is a seductive narrative because it casts the innovator in the role of the hero fighting the system and because it offers a simple explanation for failure: it wasn't my idea; it was the system. However, like all simple narratives, this one hides a more complex truth—and, for those who truly want to transform healthcare, a more useful one. Regulation is not an arbitrary obstacle, but the collective expression of something medicine has learned the hard way over centuries: that before testing something on people, one must ensure it won't kill them.

This chapter presents the second part of the book, which is devoted entirely to regulation and quality, because understanding this regulatory framework is a necessary condition for building healthcare software worthy of name. The goal is not to turn the reader into a legal expert but to equip them with a compass to navigate a territory that, although seemingly arid, holds the accumulated wisdom of decades of mistakes and successes. We explore why regulation exists, what problems it solves, what philosophy underpins it, and how to approach it not as a burden but as a tool for building better. In addition, we will do so by starting with the most fundamental question: why on earth does any of this matter?

## 2.1 WHY REGULATION MATTERS (AND WHY IT IS NOT EMPTY BUREAUCRACY)

To understand why regulation exists in the healthcare sector, it is helpful to conduct a thought experiment. Imagine that a friend, with the best intentions in the world, develops an app to help patients with diabetes calculate their insulin doses.

He's a programmer; he's spoken with some doctors, read up on the disease, and is convinced that his tool can improve the lives of millions of people. He uploads it to app stores and promotes it on social media, and within a few months, thousands of users use it daily to decide how much insulin to inject. One day, a coding error causes the app to recommend a dose ten times higher than necessary. Several patients suffer from severe hypoglycemia, some of whom end up in the emergency room; another, the most unfortunate one, does not make it in time. Who is responsible? The programmer, who did what he believed was right? The patients, who trusted an app without medical support? The app stores, which distributed it without any oversight? Do the doctors who failed to warn their patients of the risks?

This scenario, which unfortunately is not hypothetical, perfectly illustrates the need for regulation. Not because programmers are irresponsible but because medical software, like any other medical device, can cause serious harm when it fails. Unlike other fields where errors have limited consequences, here, what is at stake is people's health and lives. Regulation exists to establish a framework that minimizes the likelihood of these failures occurring and, when they do occur, to ensure clarity regarding liabilities and mechanisms for redress. It is not a bureaucratic whim but a societal response to a real problem: how to benefit from technology without exposing ourselves to unacceptable risks.

The history of healthcare regulation is, to a large extent, the history of preventable disasters that led society to say “never again.” Thalidomide in the 1960s, a drug that caused birth defects in newborns because it had not been sufficiently tested on pregnant women, gave rise to modern clinical trials.<sup>(1)</sup> The THERAC-25 disaster we observed in the previous chapter, with its lethal radiation caused by software errors, spurred specific regulations for programmable medical devices. Every tragedy has left its mark on the regulatory framework, not out of a desire to make life difficult for innovators but out of a collective determination to ensure that it never happens again. When a developer complains about how strict the regulations are, it is worth reminding them that they are written in the blood of those who paid the price for deregulation.

### 2.1.1 THE ANATOMY OF RISK: WHY MEDICAL SOFTWARE IS DIFFERENT

Some argue that this comparison is exaggerated, that software is not like drugs, that the risks are lower, and that regulation should be lighter. However, this stance ignores an uncomfortable

reality: medical software has characteristics that make it especially dangerous when it fails. It is opaque because its internal mechanisms are not visible to those who use it. It is complex because it interacts with other systems in unpredictable ways. It is modifiable because it can be updated without the user being aware of the changes. It is ubiquitous because it can be distributed to millions of people in a matter of hours. This combination of factors means that software errors can have massive reach and be extremely difficult to detect before they cause harm. Regulation is not an overreaction but a proportionate response to real risk.

The opacity of medical software presents unique challenges for error detection. When a physical device fails, the signs are usually obvious: abnormal noise, a visibly damaged component, and a function that stops responding. Software, however, can fail in completely invisible ways, producing incorrect results that seem reasonable at first glance. A diagnostic algorithm that introduces a systematic bias against a certain demographic group does not set off alarm bells; it simply produces erroneous diagnoses that can be attributed to other factors. The complexity of modern systems, with millions of interconnected lines of code, means that even their creators cannot fully predict their behavior in all circumstances. Regulation attempts to mitigate this opacity by requiring documentation, traceability, and independent verification mechanisms.

The ability to continuously modify software, seemingly an advantage, introduces stability risks that do not exist in physical devices. A mechanical infusion pump functions in the same way throughout its entire lifespan, resulting in physical wear and tear. Infusion pump control software can radically change its behavior with a remote update, potentially introducing errors that were not present in previous versions. This mutability requires change control mechanisms that assess the impact of each modification, regression testing processes that verify that fixes have not broken previous functionality, and postmarket surveillance systems that detect emerging issues. Modern regulation seeks to adapt to this reality with configuration management frameworks that treat software as a living product, not as a finished artifact.

Digital ubiquity allows errors to spread at unprecedented speed and scale. An error in a defective batch of drugs affects patients receiving that specific batch. An error in a popular health app can affect millions of users simultaneously and distributed globally before anyone is aware of the problem. This characteristic demands early detection mechanisms, rapid alert systems, and the capacity for mass recall or correction—which regulation seeks to establish. Responsibility in this context cannot fall solely on the end user, who lacks the technical capacity to assess the safety of the software they use; it must be distributed among developers, distributors, and regulators in a way that creates incentives for prevention.

### 2.1.2 THE FIVE PILLARS OF HEALTH REGULATION

It is also important to distinguish between different types of regulation because not all have the same origin or the same justification. Each of these layers has its own logic and its own history, and confusing them leads to fruitless debates where arguments that should be addressed separately are mixed together.

Safety regulations constitute the most fundamental and least controversial pillar. They stipulate that a medical device must not cause harm when used as intended and that residual risks must be acceptable in relation to the expected benefits. In the context of software, this translates into requirements for reliability, error handling, protection against misuse, and redundancy in critical functions. The IEC 62304 standard, which we discuss in later chapters, establishes a medical software lifecycle framework that prioritizes safety at every stage of development. This regulation is inherently preventive: it aims to eliminate or mitigate hazards before the product reaches the patient.

**Efficacy regulation** addresses a different but equally legitimate concern: that public and private healthcare resources are not spent on products that do not work. Software that promises to improve the diagnosis of skin cancer but, in practice, is no better than the visual assessment of an experienced dermatologist represents a waste of resources and false hope for patients. Efficacy must be demonstrated through properly designed studies, with relevant endpoints, representative populations, and rigorous statistical analyses. This regulation protects both patients and healthcare systems from irrational spending on useless or overrated technologies.

**Quality regulation** extends beyond the individual product to encompass the organizational processes that produce it. The ISO 13485 standard, applicable to quality management systems for medical devices, requires companies to maintain documented procedures for all critical activities, from supplier management to change control and from staff training to complaint management

and postmarket surveillance. The underlying logic is that safe and effective products cannot be produced consistently on a basis without an organization that systematically ensures this. This process regulation is often the most burdensome for developers, but experience shows that omitting it leads to inconsistencies that eventually result in failures.

**Privacy** and data protection **regulations**, embodied in Europe by the GDPR and in the United States by the HIPAA, among other regulations, recognize that health information is particularly sensitive. Medical data can reveal stigmatizing conditions, genetic predispositions, and behaviors that the patient would prefer to keep confidential. Its unauthorized disclosure can cause psychological, social, and economic harm that persists long after the underlying medical condition has been resolved. Regulations in this area establish patients' rights over their data, obligations for those who process it, and limits on the secondary uses that can be made of health information.

**Market regulation** ensures that competition among healthcare products occurs under conditions of transparency and fairness. It establishes labeling requirements that enable informed comparisons, prohibits deceptive commercial practices, and creates oversight mechanisms that are removed from the market products found not to deliver on their promises. This regulation is less visible to the general public but essential for the functioning of a healthcare market that is not swayed by empty marketing claims.

A common mistake among critics of regulation is to treat it as a monolithic block; in reality, it is a set of tools with varying degrees of justification. It is not the same to require that diagnostic imaging software have a minimum accuracy proven in clinical studies, as it is to require that a hospital's appointment management system meet certain quality standards. The former directly affects patients' health; the latter, primarily administrative efficiency. Lumping them together and saying "regulation is a problem" prevents us from distinguishing where it is necessary and where it might be excessive. A mature stance toward regulation consists neither of accepting it all uncritically nor of rejecting it wholesale but rather of understanding its foundations so that we can participate in its improvement.

### 2.1.3 THE ECONOMICS OF REGULATION: PROTECTING THOSE WHO DO THINGS RIGHT

There is also an economic dimension to regulation that deserves attention. In an unregulated market, medical devices compete on price and visible features but not necessarily on safety or efficacy. An unscrupulous developer could launch an unsafe but inexpensive product, and consumers who have no way of assessing safety might choose it simply because of the price. This creates what economists call a "race to the bottom," where safe but more expensive products are driven out of the market by unsafe but cheap ones. Regulation establishes a minimum quality standard that prevents this perverse dynamic and protects those who do things right from unfair competition by those who do them wrong. It is far from being an obstacle to quality innovation; it is its best ally.

The "race to the bottom" argument illustrates why pure self-regulation is insufficient in the healthcare sector. Responsible companies might try to differentiate themselves through voluntary certifications or proactive transparency, but in an asymmetric market where consumers cannot assess technical safety, these signals are easily imitated by unscrupulous competitors. Only a regulatory framework with independent verification mechanisms and credible sanctions can resolve this collective action dilemma, where what is rational for each individual actor leads to a socially undesirable outcome. Regul—is not paternalism toward irrational consumers but rather a recognition that the technical information needed to evaluate medical devices is inherently asymmetric.

Another common argument against regulation is that it stifles innovation and delays the arrival of treatments that could save lives. There is some truth to this claim: regulatory processes undoubtedly add time and cost to the development of medical devices. However, what this view overlooks is that regulation also accelerates the adoption of safe technologies by building trust among physicians, patients, and payers. A product that has undergone rigorous regulatory evaluation is far more likely to be adopted by healthcare systems and prescribed by doctors than one that has not. The speed of market entry is lost when the product reaches the market. Above all, it prevents the most tragic scenario: that of products that reach the market quickly but kill people, causing a decline in public trust that affects the entire sector.

The relationship between innovation and regulation need not be antagonistic, and in fact, countries with more developed regulatory systems tend to produce the most healthcare

innovation. This is no surprise: predictable, well-designed regulation provides a framework of certainty that allows innovators to plan for the long term and investors to risk their capital with some assurance. What kills innovation is not regulation but regulatory uncertainty: not knowing what to expect, rules changing without warning, and evaluation criteria that are opaque or arbitrary. That is why experienced developers do not call for less regulation but for clear, stable, and proportionate regulation. They know that this is the environment where they can build sustainable businesses and products that truly reach patients.

The empirical evidence on the impact of regulation on innovation is more nuanced than ideological debates suggest. Comparative studies across jurisdictions with different levels of health regulation do not show a simple correlation between less regulation and more innovation. What does consistently emerge is that the quality of regulation—measured by predictability, transparency, and responsiveness—matters more than its quantity does. Regulatory systems that have implemented fast-track pathways for products addressing high unmet needs, that maintain ongoing dialog with the industry and that update their frameworks to incorporate scientific advances, succeed in combining patient protection with the promotion of innovation. Smart regulation is a sophisticated public policy tool, not a mere bureaucratic obstacle.

#### 2.1.4 PROCESS VERSUS OUTCOME: REGULATORY BALANCE

A useful distinction for navigating this debate is the one that separates “process” regulation from “outcome” regulation. The former establishes how things should be done: which development methods to use, what documentation to generate, and what controls to implement. The latter focuses on what the product must demonstrate: that it is safe, that it is effective, and that its benefits outweigh its risks. Developers tend to complain more about the former than about the latter because process regulation imposes ways of working that they may perceive as rigid or alien to their culture. However, experience shows that both are necessary: without processing, it is difficult to ensure that results are reproducible; without results, the process becomes an empty exercise. The balance between the two is the subject of ongoing debate, and modern regulatory frameworks seek to find the optimal point.

Process regulation in medical software, embodied in standards such as IEC 62304, has evolved to recognize the unique characteristics of software development. Unlike the manufacturing of physical devices, where the production process is relatively standardized, software development allows for multiple valid methodologies, from waterfall models to agile development and from extreme programming to formal software engineering. Modern regulation aims to be agnostic regarding specific methodologies, focusing instead on the outcomes of the process: requirements traceability, configuration control, risk management, verification, and validation. This methodological flexibility, within a framework of output requirements, represents an advance over more rigid regulations that imposed specific processes incompatible with the reality of contemporary software development.

However, methodological flexibility does not mean a lack of rigor. Medical software development processes must be documented, measurable, and auditable. It must be possible to reconstruct why certain design decisions were made, how it was verified that the code correctly implemented the requirements, what tests were performed, and with what results. This documentation is not an end in itself but a means to enable incident investigation, continuous improvement, and knowledge transfer when development teams change. The tension between agility and documentation is real, but it can be resolved through tools and practices that integrate documentation generation into the workflow rather than adding it as a subsequent burden.

For its part, outcome-based regulation faces the challenge of defining what evidence is sufficient to demonstrate safety and effectiveness in a field as diverse as medical software. With respect to diagnostic software, this may involve studies comparing results with those of human experts, ROC curve analyses, and evaluations under real-world conditions. For treatment software, comparative clinical trials against current standards of care may be needed. For information management software, the evidence may focus on usability, interoperability, and data security. The diversity of products makes excessive standardization difficult, but it also creates inconsistency if each evaluator applies different criteria. Regulators attempt to manage this tension through product-category-specific guidelines while maintaining common general principles.

For the physician reading this book, understanding the logic of regulation is also a way to protect their own practices. When a healthcare professional uses a product that has not passed the appropriate regulatory checks, they are assuming a risk that is not theirs to bear. If something

goes wrong, liability may fall on them for having used an unvalidated tool. Physicians have the right to demand that the products they use have the proper authorizations, and they should be wary of those that circumvent them with arguments of innovation or agility. Regulation is not only a safeguard for patients but also for the professionals who care for them.

The involvement of clinicians in defining and improving regulation is essential, yet it is rarer than it should be. Doctors who have seen first-hand what happens when technology fails—who have had to explain to a family why a software error contributed to the death of their loved one—have a perspective that regulators and industry representatives cannot replace. Their voices should be heard on the committees that design the rules, in the agencies that enforce them, and in public debates about reform. Health care regulation is too important to be left solely in the hands of bureaucrats and business; it needs the perspective of those who have dedicated their lives to caring for patients and know what truly matters when theory meets practice.

In the specific realm of medical software, regulation poses unique challenges that set it apart from other medical devices. Software can be continuously updated, raising the question of when an update is minor and when it is so substantial that it should require a new authorization. Software can be distributed globally in a matter of minutes, challenging regulatory models based on national borders. Software increasingly incorporates artificial intelligence, which learns and changes with use, calling into question the very idea of a “finished” product that is evaluated once and is done. Regulators around the world are grappling with these issues, and the answers they provide in the coming years will shape the future of digital health.

For developers new to this field, the advice could not be clearer: do not view regulation as a formality to be addressed at the end, once the product is already finished. That is a sure path to failure, because you will then discover that there are fundamental aspects you should have considered from the start and that modifying them is now impossible or prohibitively expensive. Regulation must be integrated into the development process from the very first sketch, as just another design variable, as a constraint that, when properly understood, can even stimulate creativity. The best healthcare products are not those that circumvent regulation but those that incorporate it as part of their value proposition.

It is also worth debunking the myth that regulation is something that matters only in wealthy countries and that in developing countries, it can be ignored without consequences. This view, in addition to being paternalistic, is profoundly mistaken. Patients worldwide deserve the same protection against unsafe products, and the healthcare systems of countries with fewer resources are, if anything, more vulnerable to the harm caused by defective technology. Furthermore, ignoring regulation closes the door to international collaboration and funding because investors and global partners demand minimum standards. Regulation is not a luxury for wealthy countries but a prerequisite for participation in the global digital health market.

To conclude, this introduction to the logic of regulation, let us return to the thought experiment from the beginning. Let's now imagine that our developer friend, instead of simply launching his app, has taken the time to understand the applicable regulatory framework. He has correctly classified his product as a Class IIb medical device, implemented a quality system in accordance with ISO 13485, documented the development process in accordance with IEC 62304, conducted risk management in accordance with ISO 14971, and subjected the product to a clinical evaluation demonstrating its safety and effectiveness. When their app reaches the market, it won't be perfect—because nothing is—but it will be far less likely to contain serious errors. If an incident does occur despite everything, there will be traceability to investigate it, processes to correct it, and clarity regarding responsibilities. That product, even if it took longer to arrive, is infinitely preferable to the first one, both for patients and for the developer themselves, who will be able to sleep at night without the nightmare of having caused preventable harm.

Regulation matters because medicine matters. This matters because patients believe that the products they use have been rigorously evaluated. Doctors need reliable tools to practice their profession. It matters because society has decided, after many painful lessons, that health is not just another consumer good but also a right that deserves special protection. Understanding this is not an act of faith but the first step toward building healthcare technology that truly deserves that name. It is also the gateway to the more concrete details we explore in the coming chapters: the specific standards that translate this philosophy into tangible requirements, the processes that turn principles into practice, and the tools that allow us to navigate the regulatory maze without being lost in it.

## 2.2 TRACEABILITY FOR DEVELOPERS ACCUSTOMED TO JIRA

“Traceability isn't for the present; it's for the future, and the future is unpredictable.”

Let's imagine for a moment the everyday scene at a conventional software startup. The team uses Jira, or a similar tool, to manage their work. Each task is represented by a card that moves through different columns: pending, in development, in testing, completed. Developers create branches in the code repository, link their commits to the corresponding tasks, and when everything is ready, they merge the changes and close the issue. It is an efficient, flexible workflow that has proven its value in thousands of successful projects. No one questions whether this way of working is suitable for software development in general. The problem arises when this same team, with this same culture, faces a medical software project for the first time, and someone tells them that they need “traceability.” Suddenly, what was once simple and agile becomes complex and bureaucratic. Developers feel trapped in a maze of documents, records, and requirements that they do not understand and perceive as a hindrance to their creativity. However, traceability is not a whim of regulators but a necessity stemming from something we have already explored: in medicine, errors have consequences we cannot afford.

Traceability is the ability to reconstruct the path from a clinical need to the code that implements it and from a piece of code to the need that justifies it. In other words, it is the ability to answer questions such as the following: Why was this line of code written? What requirement was it fulfilling? Who decided it? When was it done? What tests verify that it works correctly? Conversely, given a clinical requirement, where is it implemented in the code? How can I ensure it is met? What would happen if I modify it? This ability to trace design and development decisions in both directions is what distinguishes medical software from a conventional application, and it is also what allows for investigating incidents, correcting errors, and demonstrating to a regulator that the product has been developed with the necessary rigor.

For a developer accustomed to Jira, this concept may not be entirely unfamiliar. After all, in many projects, commits are linked to tasks, which is already a rudimentary form of traceability. The difference lies in the level of rigor and purpose. In a conventional project, the relationship between tasks and code is useful for understanding what was done, but it is rarely required for audit or security purposes. If a developer forgets to link a commit to a task, the worst that can happen is that someone has to dig a little deeper to understand the change. In a medical project, the same omission can have serious consequences when, months later, an incident forces a review of the history, and it cannot be determined why a modification was introduced that turned out to be dangerous. Medical traceability is not an organizational option but a safety requirement.

### 2.2.1 FROM JIRA TO BIDIRECTIONAL TRACEABILITY: EXPANDING THE MODEL

The standard for establishing traceability requirements for medical software is IEC 62304, which we explore in detail in the next chapter. For now, suffice it to say that this standard requires a bidirectional relationship between requirements, design, code, testing, and identified risks. This means that for every requirement, there must be a code that implements it and tests that verify it. For every piece of code, there must be a requirement that justifies it. For every identified risk, control measures must be implemented and verified. This entire network of relationships must be documented in a way that allows it to be audited by a third party. It is not about generating mountains of meaningless paperwork but about building a network of connections that allows us to navigate the system safely and understand the implications of any change.

Developers accustomed to Jira often have a visceral reaction to this approach: “that's impossible to maintain,” “it will slow us down enormously,” “it's useless bureaucracy.” It is true that if approached naively, traceability can become a paralyzing burden. However, the experience of teams that have been working on medical software for years shows that it is possible to integrate these requirements into agile workflows, provided that the right tools are used and, above all, that the purpose of what is being done is understood. The key is not to treat traceability as an afterthought but as another dimension of development that is incorporated from the start and automated as much as possible.

A tool such as Jira can serve as an excellent foundation for traceability, provided that it is configured properly. The idea is that every requirement, every risk, every test case, and every development task is a traceable item with a unique identifier. The relationships between these items are established through links: “implements,” “verifies,” “mitigates,” and “depends on”. When

a developer writes code, they associate their commits not only with the development task but also with the requirements that gave rise to it and the risk it addresses. When a tester runs tests, they associate the results with the test case and, through it, with the corresponding requirements. In this way, the tool becomes a living repository of all the relationships that will later allow the product's history to be reconstructed.

The practical implementation of this bidirectional traceability requires a richer information structure than typical task management does. While in conventional Jira, a user story may be linked to several development and testing subtasks, in the medical model, that story must also connect to the derived system requirements, the test cases that verify it, the risks it mitigates, and the code that implements it. This network of relationships is not built automatically; it requires discipline in creating links and in the naming of identifiers. However, modern application lifecycle management (ALM) tools can automate much of this management, extracting relationships from code repositories, testing systems, and risk management tools.

The real challenge is not technical but cultural. For many developers, the idea of exhaustively documenting what they do goes against their training and professional identity. They have been educated in the culture of “self-explanatory code” and “minimum necessary documentation.” They have seen too many projects where documentation becomes an end in itself, disconnected from the reality of the code. They rightly distrust any process that could turn into an empty bureaucratic exercise. The challenge for leaders of medical software projects is precisely to demonstrate that traceability does not have to be that way that it can be naturally integrated into the workflow and that, far from being a burden, it provides tangible benefits for product quality and the team's peace of mind.

## 2.2.2 HIDDEN BENEFITS: WHY TRACEABILITY MAKES US BETTER

One of these benefits is the ability to assess the impact of changes, which is critical in medical software. When a developer modifies a line of code, they need to know which requirements may be affected, which risks might be reintroduced, and which tests would need to be repeated. In a project without traceability, this assessment relies on people's memories, which are fragile and unreliable. In a project with good traceability, the system itself can help identify all the elements related to that piece of code, facilitating a systematic assessment and reducing the risk of overlooking something important. What appears to be a regulatory requirement thus becomes a tool for working better.

Change impact analysis is particularly valuable in the maintenance of medical software, which can span decades. When the need arises to fix a bug or adapt the system to a new regulatory requirement, the developer must be able to determine precisely which components will be affected. Without traceability, this determination requires exhaustive code reading and tacit knowledge that may have been lost because of staff turnover. With traceability, documented relationships allow for quickly identifying the scope of the change, estimating the required effort, and planning the corresponding regression tests. This capability reduces the incident response time and the likelihood that fixes introduce new bugs.

Another less obvious but equally important benefit is the protection of the team itself. In the event—always a possibility—that a serious incident occurs, authorities will investigate what happened and why. If the team can demonstrate that it followed established processes, that it properly documented decisions and that tests confirmed correct functionality, its position will be much stronger than if everything depends on memory and goodwill. Traceability is not only a requirement of regulators but also a shield for developers against liabilities that do not belong to them. In an environment where errors can end up in court, having a reliable record of what was done and why it was done is a form of self-protection.

Traceability also facilitates communication with external stakeholders who do not have direct access to the code. Regulators, auditors, and hospital clients need to understand what the system does without reading lines of code. The traceability chain, from high-level requirements to executed test cases, provides a comprehensible narrative of the product that can be reviewed and questioned. This transparency is not just about regulatory compliance; it is also trust-building marketing, a demonstration of professionalism that distinguishes serious providers from opportunists. In a market where differentiation based on technical features is becoming increasingly difficult, the ability to demonstrate robust processes becomes a competitive advantage.

For developers coming from the world of conventional software, one of the most difficult

concepts to grasp is that of “freezing” or “baseline.” In typical agile projects, the code is in a state of constant flux, and that is considered a virtue. In medical software, however, every version that is released must be perfectly identified and frozen so that it is possible to reconstruct exactly which code was running at any given moment. This does not mean that iterative development cannot continue, but it does mean that every iteration that reaches production must be treated as a unique and immutable artifact. Commits to that version after it has been released make no sense; any change must result in a new version, with its own identifier and its own documentation.

This requirement clashes with the common practice of many teams, which continuously deploy small improvements with little documentation. However, it has a powerful rationale: when a patient suffers harm because of a software error, it is crucial to be able to determine exactly which version was in use, to investigate whether the error is present in all versions or only in some, and to be able to notify the centers using affected versions. Without this ability to identify versions precisely, the response to an incident descends into chaos, and patient safety suffers. Version immutability is not a bureaucratic whim but an epidemiological safety requirement.

Version management in medical software requires formal configuration control procedures that go beyond standard code versioning. There must be a record of which versions of each component are included in each product release, what changes were made compared to the previous version, and what risk management decisions justified those changes. This record, known as a **Design History File** in regulatory terminology, is a living document that accompanies the product throughout its entire lifecycle. Maintaining it requires discipline, but it provides the basis for informed decisions regarding updates, market withdrawals, and safety communications.

### 2.2.3 ADAPTING AGILE PRACTICES: BRANCHES, COMMITS, AND DOCUMENTATION

Another common practice in agile development that must be adapted is that of code branches. In conventional projects, it is common to have multiple development branches, merge them frequently, and sometimes even rewrite the repository history to keep it clean. In medical software, these practices are problematic because they hinder traceability. If the history is rewritten, the ability to know exactly what code was present at any given time is lost. If branches are merged without rigorous control, it becomes difficult to associate each change with its requirements and tests. This does not mean that branches cannot be used but rather that they must be managed with greater discipline, maintaining a clear record of which changes were included in each version and why.

The branching strategy in medical software must balance the need for parallel development with the requirement for traceability. The feature branches are acceptable, but they should be shortlived, and their merging must be documented with references to the corresponding requirements and tests. Release branches allow versions to be stabilized while development continues, but they must be protected against unauthorized modifications. The practice of “force push,” which rewrites the repository history, must be prohibited on main branches to preserve the integrity of the historical record. These restrictions do not prevent agile work, but they require that agility be exercised within a framework of documentation discipline.

Documenting design decisions is another aspect that often clashes with developer culture. In many teams, decisions are made in conversations, discussed in meetings, and rarely recorded in writing. It is assumed that the code reflects the final decision—and that this is sufficient. In medical software, this is not acceptable because the code does not explain why one decision was made over another. When, months later, someone wonders why a feature was implemented in a certain way, the answer is not in the code but in the reasoning that led to that solution. Recording that reasoning, even succinctly, is essential for maintaining traceability and facilitating long-term maintenance.

Architecture decision records (ADRs) are agile practices that are particularly well-suited to the needs of medical software. Every significant architectural decision is documented in a standard format that includes the context, the decision made, the alternatives considered and rejected, and the expected consequences. These documents, stored alongside the code in the repository, provide decision traceability without excessive documentation overhead. Maintaining them is part of the normal workflow, not an added task, and their structured format facilitates review by auditors and regulators. Adopting ADRs represents a practical compromise between agility and rigor.

Some argue that all this documentation is unnecessary when the team is small and when communication is fluid. “We talk to each other; we know what we did and why.” This argument,

understandable from the perspective of a tightly knit team, ignores several realities. The first is that teams change: people leave, new people arrive, and what was obvious to the founders becomes a mystery for newcomers. The second is that incidents can occur years later, when no one remembers the details anymore. Third, regulators do not accept “ask Maria” as evidence that something was done correctly. Traceability is not for the present; it is for the future, and the future is unpredictable.

#### 2.2.4 TOOLS, MATURITY, AND THE CULTURE OF CARE

To facilitate the adoption of these practices, teams should use tools that naturally integrate traceability. There are requirements management systems that integrate with Jira, code repositories that allow commits to be linked to items in those tools, and testing environments that automatically record the relationships between test cases and results. Investing in this infrastructure is not an expense but an investment in future productivity because it reduces the manual effort required to maintain traceability and minimizes the risk of errors due to omission. Compared with a conventional team dedicated to its usual tools, a well-equipped team can maintain traceability with little more effort.

The application lifecycle management (ALM) tool ecosystem has evolved significantly to support the needs of medical software. Platforms such as Jama Connect, Polarion, or DOORS Next Generation enable the management of requirements with full traceability, integrating with test management systems and code repositories.<sup>(2)</sup> These tools automate the generation of traceability matrices, identify orphaned elements without links, and facilitate the generation of reports for audits. Their adoption requires investment and training, but the return in terms of reduced manual effort and improved quality is substantial. For startups with limited resources, there are also open-source solutions and configurations of generic tools that provide basic traceability functionality.

The developer's attitude toward traceability is also a matter of professional maturity. Early in their careers, many programmers view documentation as a waste of time that prevents them from writing code, which is what they truly enjoy. With experience, they learn that code is only a small part of the software lifecycle and that decisions made during development have consequences that extend far beyond the moment they are written. They learn that well-documented code is easier to maintain, that a clear history of decisions saves hours of research and that traceability is not an end in itself but a means to build more reliable and safer software. This maturity is especially necessary in medical software, where the consequences of immaturity can be tragic.

For teams taking their first steps in this field, a useful strategy is to start with a basic level of traceability and gradually increase it. It is not about implementing a perfect system from day one that covers all aspects of the standard but rather about identifying the critical elements and securing them first. For example, ensuring that every requirement has associated tests and that every code change can be traced back to a requirement or a documented bug fix. From there, complexity can be added as the team becomes familiar with the processes and tools. What matters is not initial perfection but the direction of the journey: toward a culture where traceability is second nature, not an external imposition.

Gradual implementation allows the team to learn from its mistakes without catastrophic consequences. Traceability maturity milestones can be established: first, that all code is associated with an issue tracking ticket; then, that every ticket is linked to a requirement; subsequently, that verifiable test cases exist for each requirement; and finally, that risks are identified and mitigations are verified. Each milestone achieved provides immediate value to the project, regardless of future regulatory audits. This incremental approach is more sustainable than attempting to implement the entire traceability framework all at once, which often collapses under its own complexity.

There is also a psychological dimension to traceability that deserves attention. For many developers, the idea that every action is logged creates a sense of surveillance that can be uncomfortable. It is important for project leaders to clearly communicate that traceability is not a control mechanism but a tool for collective protection. It is not about monitoring each person's work but about building a shared record that benefits everyone. When an error occurs, traceability allows us to understand what went wrong without needing to assign individual blame, focusing attention on improving the system rather than punishing individuals. Used properly, traceability fosters a culture of fairness and learning, not a culture of fear.

Traceability is not an end in itself but a means to a higher end: patient safety. When a developer is tempted to skip a step, to not document a decision, to forget to link a commit to its

requirements, they should ask themselves: What would happen if this small oversight contributed to an error that harmed someone? Could I forgive myself? Could I explain to a patient or their family why I didn't take five minutes to make things clear? These uncomfortable but necessary questions constitute the true foundation of traceability. It is not the standard; it is not the regulator; it is not the boss. It is the awareness that what we build can save lives or harm them and that we have a responsibility to do everything in our power to tip the scales toward the former.

The transition from a Jira culture to a traceability culture is not easy, but it is possible and necessary. It requires changing habits, incorporating new tools, and adopting a more disciplined mindset. However, it also offers rewards: the peace of mind that comes from knowing what you build is solid, the ability to sleep at night without fear of what might go wrong, and the pride of belonging to a profession that takes its responsibilities seriously. For developers who decide to take this step, medical software ceases to be a source of frustration and becomes a fascinating field where technology and ethics meet, where every line of code has a clear purpose, and where a job well done has a direct impact on people's lives. It's worth the effort.

## 2.3 WHAT IS A DESIGN HISTORY FILE (DHF) AND WHY IT'S YOUR ALLY?

Let's imagine we're building a house. Not just any house, but the house where we'll live the rest of our lives, where we'll raise our children, where we hope to grow old. During construction, we make decisions about the foundation, materials, electrical work, and plumbing. We choose a type of brick, a brand of window, and a model of a boiler. We hire architects, quantity surveyors, masons, and electricians. Months go by, the house is finished, and we move in. Twenty years later, a pipe bursts and causes a flood. The plumber that comes to fix it needs to know what type of pipe was installed, where the pipes run, and what pressure they can withstand. If there are no plans, if there are no records, if everything depends on the memory of those who are no longer here, the repair becomes a nightmare. If what fails is not a pipe but something that could kill someone, the nightmare multiplies. The Design History File is, exactly, the set of plans and records for that house but is applied to medical software.

Design history files, known by their acronym DHF, are central concepts in medical device regulation, and yet, they are among the concepts that generate the most confusion among developers new to this field. The formal definition, as it appears in the regulations, states that the DHF is the set of records describing the design history of a finished medical device. In simpler terms, it is the repository for all the information needed to understand how the final product was developed: from the initial ideas and requirements, through detailed drawings, to meeting minutes where decisions were made, test results, risk analyses, and any other document that was significant during the development process. It is not a single document but an organized collection of documents—a living archive that grows and evolves throughout the product's lifecycle.

For a developer accustomed to conventional software, the idea of maintaining such a comprehensive archive can be overwhelming. In their previous experience, documentation is often sparse, scattered, and limited to what is strictly necessary for the team to understand each other. Requirements live in Jira, the design in the developers' heads, decisions in hallway conversations, and tests in scripts that are run and forgotten. This way of working is efficient for certain types of projects, but it is completely inadequate for medical software because it does not allow the process to be reconstructed when something goes wrong, nor does it allow one to demonstrate to a regulator that everything possible was done to ensure safety. The DHF fills that gap, providing a framework for organizing information in a way that is useful not only during development but also throughout the product's entire lifecycle.

### 2.3.1 THE ANATOMY OF THE DHF: WHAT IT CONTAINS AND WHY

The first question that arises regarding the DHF is as follows: exactly what documents should it contain? The answer depends on the type of device and its classification, but every regulatory authority expects to find a common core.

First, the design requirements include what the product must do, with what precision, and under what conditions. Second, the design drawings include how the software is organized, what components make it up, and how they relate to one another. Third, we discuss traceability in the previous section: which requirements each component implements and which tests verify

each requirement. Fourth, the risk analysis addresses what can go wrong, with what probability, with what consequences, and what has been done to prevent or mitigate it. Fifth, the test results included what was tested, how, with what results, and what decisions were made on the basis of them. In addition, sixth, the change log: what modifications have been made throughout the product's lifecycle, why, and who authorized them.

This list may seem long, but it is important to understand that these are not separate, disconnected documents but rather a network of information where each piece is interconnected with the others. The requirements justify the design. The design determines the risks. These risks necessitate testing. The tests verify the requirements. The changes affect all of the above. The DHF is precisely the tool that keeps this network coherent, ensuring that when something is modified, all its implications are reviewed and that when someone asks why something was done, they can find the answer. It is not a dead archive but a living organism that reflects the current state of knowledge about the product.

The physical structure of the DHF can be organized in many ways, from hierarchical folders in a document management system to specialized databases. The key is to have a navigation logic that allows any information to be found quickly and for documents to be uniquely identified with codes indicating their nature, version, and date. A best practice is to maintain a master index, updated automatically if possible, that allows documents to be located by multiple criteria: by development phase, by document type, by functional area, and by date. This structure should not be rigid; it must allow for the incorporation of new document categories that emerge with experience.

### 2.3.2 DHF AND AGILE METHODOLOGIES: APPARENT CONTRADICTION, REAL COMPLEMENTARITY

For teams working with agile methodologies, the idea of maintaining a DHF may seem to contradict the philosophy of “responding to change rather than following a plan.” However, this apparent contradiction is resolved when one understands that the DHF is not a rigid, unchangeable plan but rather a record of what is being done as it is done. Instead of producing a mountain of documents at the outset and then trying to make reality fit them, the process is documented as it progresses in an iterative and adaptive manner. Each sprint, each iteration, generates new information that is incorporated into the DHF. By the end of development, the file contains the complete history of the product—not as a bureaucratic imposition but as the natural byproduct of a job well done.

This way of working does, however, require a certain amount of discipline. You can't leave all the documentation for the end, because then it would be impossible to reconstruct the process accurately. You have to document as you go, even if only briefly, and ensure that the information is organized so it can be easily found when needed. Tools such as wikis, document repositories, or even a well-configured Jira itself can serve as a foundation for the DHF, provided they are used systematically. What matters is not the platform but the existence of a central location where all relevant information is accessible and interconnected.

Integrating the DHF into agile sprints can be achieved by defining a “definition of done” that includes documentation requirements. A user story is not complete merely when the code works but when the requirement is documented in the DHF, when test cases are recorded, and when design decisions are noted. This expansion of “done” requires the team to set aside time in each sprint for these activities, explicitly estimating them rather than treating them as invisible. Over time, documentation becomes a habit, and its marginal cost is significantly reduced.

### 2.3.3 UNEXPECTED BENEFITS: KNOWLEDGE, AUDITING, AND SURVIVAL

One of the least obvious but most valuable benefits of the DHF is that it facilitates the onboarding of new team members. In conventional software development, when someone joins a project, it takes them weeks or months to understand how the system actually works because much of the knowledge resides in people's heads. In a project with a well-maintained DHF, the new member can review the product's history, understand why certain decisions were made, and learn about the identified risks and the tests performed. This greatly accelerates their onboarding and reduces the team's reliance on specific individuals. The DHF thus becomes a knowledge management tool that protects the project from staff turnover and preserves accumulated

knowledge.

Another key benefit is audit readiness. At some point, a regulator will likely want to examine the product and the process that generated it. If the team has maintained a well-organized and complete DHF, the audit will be a relatively straightforward process: simply present the information, explain how it is organized, and answer any questions that arise. If, on the other hand, the documentation is scattered, incomplete, or simply nonexistent, the audit will become a traumatic experience, with the added risk that the regulator may demand additional studies or even halt the product's commercialization. The DHF is not a favor for the regulator but an investment in future peace of mind.

Some people believe that DHF is something that matters only to large companies with the resources to maintain quality and documentation teams. This idea is mistaken and dangerous. It is precisely startups and small businesses that need good DHF the most because they have the least margin for error and suffer the most from the consequences of poor knowledge management. A small team that loses its lead developer without having properly documented the product may be forced to shut down. A team that has maintained an organized DHF can survive that loss and continue development with new staff. Far from being a luxury for large corporations, the DHF is a survival tool for any organization that builds medical software.

The relationship between the DHF and the source code deserves separate consideration. The code is, without a doubt, the most precise expression of what the product actually does. However, the code does not explain why it was implemented in a certain way, what alternatives were considered, or what evidence supports its correctness. The DHF complements the code by providing the context that the code cannot contain. Together, code and DHF form a whole that allows for an in-depth understanding of the product: the code says what it does, the DHF says why it does it and how we know it does it right. Separately, both are incomplete. This complementarity is important to understand because developers, in their love for code, often tend to underestimate the importance of documentation. Code is necessary, but it is not sufficient.

### 2.3.4 DHF IN PRACTICE: STRUCTURE, LEVEL OF DETAIL, AND CHANGE MANAGEMENT

A practical question that often arises is how much detail is enough. The answer, as with so many things, is: it depends. It depends on the product's complexity, the associated risks, and the applicable regulatory requirements. However, there is a guiding principle: the documentation must be sufficient for a competent person, with no prior knowledge of the product, to understand how it was developed and why the decisions that were made were made. This does not mean that every thought must be documented, but it does mean that the fundamental milestones of the process must be recorded. A good exercise is to ask yourself: if I were to leave the project tomorrow and someone else had to continue it, what information would they need? That is, roughly speaking, the information that should be included.

For teams creating a DHF for the first time, it can be helpful to start with a simple structure and refine it as you gain experience. A possible initial structure could include folders or sections for requirements, architecture, risks, testing, changes, and key decisions. Within each section, documents are organized chronologically and uniquely identified. Over time, the team will discover what additional information is needed and how best to organize it. What matters is not initial perfection but consistency and a commitment to continuous improvement. Like the product itself, it is a living artifact that evolves with the project.

Digitizing the DHF is another aspect to consider. In the past, these files were typically physical and consisted of paper folders and documents. Today, fortunately, there are tools that allow them to be maintained electronically, with all the advantages that include quick search, links between documents, version control, and remote accessibility. However, digitization also poses challenges: You must guarantee the integrity of the documents, protect them from unauthorized modifications, and ensure their long-term availability. It is not enough to simply store the documents on a shared drive; you need a system that guarantees their authenticity and traceability. Quality management tools designed specifically for regulated environments can be a good investment, although for small teams, a combination of a code repository, a wiki, and best practices may suffice.

One aspect that often surprises developers is that the DHF does not end when the product is launched. In fact, in many ways, that is when its useful life truly begins. Every change, every fix, every improvement made to the product throughout its lifecycle must be recorded in it. Every incident that occurs, every new risk assessment, and every test update must be incorporated

into the file. It is the product's logbook, and as such, it must be kept up to date for as long as the product exists. This implies that the discipline of documentation is not something that is abandoned after launch but rather forms part of the daily work of maintaining and evolving the software.

For physicians involved in development projects, understanding what the DHF is and what it is used for is also important. When a clinician collaborates in defining requirements, identifying risks, or validating tests, they contribute to the content of the DHF. Their contributions are recorded and become part of the product's history. If any questions arise in the future about why a particular feature was included or excluded, their testimony will be present. The DHF also protects their work, ensuring that their knowledge is not lost and that their contribution is recognized. That is why physicians should take an interest in how this information is organized and insist on being involved in its creation.

The relationship between DHF and intellectual property is another aspect that deserves attention. The DHF contains detailed information on how the product was built, including aspects that may be subject to patents or trade secrets. That is why it is important to manage access to the DHF carefully, ensuring that only authorized individuals can view it and that there are records of who accesses what information. Moreover, the DHF can serve as valuable evidence in intellectual property disputes, as it documents the creation process and can help establish who did what and when. Like so many other things, the DHF is a double-edged sword: well managed, it protects; poorly managed, it exposes.

To conclude, let us return to the house metaphor we started with. When we live in a house, we rarely consider the blueprints. Only when something breaks, when we want to renovate, when we need to understand how it's built—that's when we appreciate that they exist. The same is true for medical software: during development, the DHF may seem like a burden, a distraction from what matters—which is writing code that works. However, when the product has been on the market for years, when the original teams are no longer there, when unexpected problems arise, the DHF reveals itself for what it always was: the memory that allows the product to outlive its creators, to be maintained and improved, to continue fulfilling its role of helping patients without causing harm. Building a good DHF is, ultimately, an act of responsibility toward the future, a recognition that what we do today will have consequences long after we have moved on to other things. In addition, that responsibility, in the field of medicine, is the only thing that truly matters.

If the DHF is the memory that allows medical software to outlive its creators, how do we ensure that this memory is not only complete but also honest, preventing the pressure to document "correctly" from ultimately generating records that justify decisions made for other reasons after the fact, and ensuring that traceability does not become a well-written fiction?

## 2.4 ISO 13485: QUALITY, CHANGE CONTROL, AND AUDITS

"Quality, in its deepest sense, is the ability to consistently produce results that meet expectations. In addition, that ability does not reside in the code, but in the system that produces it."

For many developers, quality is synonymous with clean, well-structured code that is backed by automated tests, meets requirements, and has no obvious errors. It is a product-centric view focused on the technical artifact delivered to the user. While this view is not wrong, it is incomplete, especially in regard to medical software. In this field, quality is not just a property of the final product but also of the process that generates it. A product may be technically flawless yet have been developed in such a way that there is no guarantee that the next product—or the next version—will be as well. In its deepest sense, quality is the ability to consistently produce results that meet expectations. In addition, that ability does not lie in the code but in the system that produces it. That is exactly what ISO 13485 is about.

ISO 13485 is an international standard that specifies the requirements for a quality management system applicable to organizations that design, manufacture, install, and service medical devices.<sup>(3)</sup> It is not a standard about what the product should be like but about what the organization that creates it should be like. It establishes which processes must exist, how they must be documented, how they must be controlled, and how they must be improved. In a sense, it is a standard about the organization, not about the product. This is precisely what makes it so valuable and so misunderstood by those encountering it for the first time. Because they will not find technical specifications in it regarding what their software should be like, but rather requirements regarding how they should document their decisions, how they should manage

changes, how they should train their staff, and how they should audit themselves.

For a developer accustomed to the culture of code, this approach can be disconcerting. What does staff training have to do with software quality? Why should I document my processes if what matters is that the product works? These questions reveal a profound misunderstanding of what it means to construct safe products in a sustainable way. A product that works today may stop working tomorrow if the person who developed it leaves and leaves no records. A team that produces excellent code under pressure may not be able to replicate it when conditions change. An organization that does not learn from its mistakes is doomed to repeat them. ISO 13485 is not a set of bureaucratic hurdles but a systematic attempt to address these problems to build organizations capable of producing quality reliably and predictably.

#### 2.4.1 THE QUALITY MANAGEMENT SYSTEM: ORGANIZATION FOR TRUST

At the heart of ISO 13485 is the concept of a “quality management system.” This is simply the set of processes, procedures, and responsibilities that an organization establishes to ensure that its products consistently meet applicable requirements. This is not something abstract or theoretical but rather very concrete matters: how customer requirements are gathered, how solutions are designed, how they are verified, how they are validated, how changes are managed, how suppliers are selected, how staff are trained, how nonconformities are investigated, and how corrective actions are implemented. All these processes, when well-defined and properly executed, constitute the quality management system. ISO 13485 provides a framework for defining them in a way that is acceptable to regulators.

The implementation of a quality management system begins with management commitment. Without the explicit and visible support of top management, any quality system will fail. This involves appointing a management representative for quality with sufficient authority to enforce processes, allocating adequate resources, establishing a clear quality policy, and periodically reviewing the system to ensure its continued suitability. This commitment is not merely rhetorical: auditors verify that management actively participates in system reviews, makes data-driven decisions, and does not allow commercial pressure to compromise quality requirements.

The quality management system documentation follows a pyramid structure. At the top is the quality manual, which describes the scope of the system and the interactions between processes. Below are the documented procedures, which detail how critical activities are performed. Next are the work instructions, which specify how to perform specific tasks. Finally, the records demonstrate that the processes have been carried out. This structure is not rigid: the standard allows for flexibility in the extent of the documentation depending on the size of the organization and the complexity of its processes. Importantly, consistency across the levels is present, and each document is kept up to date.

#### 2.4.2 CHANGE CONTROL: DISCIPLINE WITHOUT RIGIDITY

One of the most relevant aspects of the standard for software development is change control. In any software project, changes are inevitable and desirable. Requirements evolve, errors are discovered, and new opportunities arise. However, in medical software, every change must be managed carefully because it can have unforeseen consequences for the safety or effectiveness of the product. ISO 13485 requires organizations to have a defined process for managing changes, including assessing their impact, verifying that they have been implemented correctly, and validating that they have not introduced new risks and documenting all of this. The goal is not to prevent changes but to ensure that they are made in a controlled and deliberate manner.

For a team accustomed to modifying code on the fly, deploying it multiple times a day, and experimenting with new features in production, this approach can be paralyzed. If applied rigidly, it can become so. However, the experience of teams that have been working in regulated environments for years shows that it is possible to maintain agility without sacrificing control. The key lies in having processes commensurate with the risk, in automating as much as possible, and in understanding that control is not an end in itself but a means to ensure that changes do not cause harm. A mature team in this environment does not view change control as an obstacle but as a safety net that allows it to innovate with confidence.

The typical change control process in an ISO 13485-certified organization includes several

well-defined steps. First, the change request describes what needs to be changed and why. Second, the impact assessment, which analyzes how the change affects requirements, design, risks, testing, and documentation. Third, review and approval by authorized personnel, who decide whether the change is appropriate and under what conditions. Fourth, the change was implemented following established development and testing processes. Fifth, verification that the change has been implemented correctly and has not introduced any unintended side effects is needed. Sixth, we update all affected documentation, including the DHF discussed in the previous section. In addition, seventh, closing out the change, with a record of everything that was done.

This process may seem excessive for minor changes, and in fact, it would be if applied in the same way to all changes. That is why the standard allows organizations to define different levels of control on the basis of risk. A change affecting a safety-critical functionality will require a thorough process; a purely cosmetic change to the user interface may have a lighter process. Importantly, the organization clearly defines the criteria for classifying changes and consistently applies the corresponding processes. Proportionality is the key to not stifling innovation with bureaucracy.

Implementing control levels requires a change classification matrix that evaluates multiple dimensions: the affected component (critical or noncritical), the type of change (functional, bug fix, or enhancement), the scope (local or global), and the deployment environment (production, testing, or development). By combining these variables, the organization can define differentiated approval workflows: trivial changes can be approved by the team itself, major changes require quality review, and critical changes require management authorization. This flexibility, formalized in written procedures, allows for maintaining agility without sacrificing security.

### 2.4.3 NONCONFORMITIES: LEARNING FROM MISTAKES

Another central aspect of ISO 13485 is the management of nonconformities. A nonconformity is any deviation from what is specified: a requirement that is not met, a process that is not followed, and a product that does not function as it should. In organizations with mature quality systems, nonconformities are neither hidden nor ignored but are systematically investigated to understand their causes and prevent them from recurring. This is among the most profound differences between traditional software culture and quality culture. In the former, errors are corrected and forgotten; in the latter, errors are investigated and turned into opportunities for improvement.

When nonconformity occurs, the typical process includes accurately describing the problem, investigating its immediate and root causes, defining corrective actions to eliminate those causes, implementing the actions, verifying that they have been effective, and documenting the entire process. This systematic approach to problem-solving is extraordinarily powerful because it prevents the same errors from recurring repeatedly. Instead of constantly putting out fires, an organization is built that learns from its experiences and becomes more robust over time. The investment in investigating the causes is worthwhile because every problem solved at its root is a problem that will not happen again.

The distinction between corrective actions (fixing the immediate problem) and corrective measures (eliminating the cause to prevent recurrence) is fundamental but often confusing. When a software error is detected, the corrective action is to fix the code. The corrective action could be to review the code review process to detect similar patterns, improve training in that technical area, or modify the static analysis tools. Both are necessary: the remedial action to resolve the current incident and the corrective action to prevent future ones. Immature organizations focus exclusively on remedial actions, leaving the underlying causes unaddressed.

Root cause analysis often uses methodologies such as cause tree analysis or the “5 Whys” technique. These tools structure the reflection to go beyond superficial explanations (“the error occurred because the developer made a mistake”) toward systemic causes (“the error occurred because the review process failed to detect the inconsistency because the workload prevented thorough reviews because planning did not account for contingencies”). Uncovering these causes can be uncomfortable because they often involve deficiencies in processes or resources, but genuine improvement can be achieved only by addressing them.

#### 2.4.4 COMPETENCE, SUPPLIERS, AND THE QUALITY CHAIN

Staff training is another ISO 13485 requirement that often surprises developers. The standard requires that individuals performing work that affects product quality be competent and that the organization maintain records of that competence. This involves defining the profiles for each position, assessing whether the individuals filling them meet the requirements, providing training when necessary, and documenting all of this. For many technicians, accustomed to having their practical experience valued over credentials, this requirement may come as a shock. However, it makes sense: if it has not been verified that a person knows what they need to know, their work cannot be trusted to meet the required quality standards. Training is not a formality but a guarantee.

Competency management goes beyond initial training. The standard requires that staff be periodically assessed to ensure that they maintain their competency, especially when there are changes in processes or regulatory requirements. This involves maintaining competency matrices that link each position to the required knowledge and skills, identifying gaps through assessments, and closing them through planned training. Training records must include content, date, instructor, and evidence of effectiveness assessment. This systematic approach ensures that organizational capability does not depend on specific individuals.

The relationship with suppliers is another important aspect of the standard. In medical software development, it is common to use third-party components: libraries, frameworks, cloud services, and development tools. All these suppliers must be evaluated to ensure that their products or services meet the necessary requirements. You cannot simply download a library from the internet and trust that it is safe; you must verify its origin, quality, support, and maintenance. If the supplier is critical to the product's safety, it may be necessary to audit their processes or require specific certifications. This requirement may seem excessive, but it reflects a reality: a failure in a third-party component can have the same consequences as a failure in the product's own code, and the responsibility still lies with the manufacturer of the final product.

Supplier evaluations must be documented and reviewed periodically. For software components, the criteria include project maturity, update frequency, the user community, vulnerability history, licensing, and support capabilities. Critical suppliers may require specific quality agreements, audits, or ISO 9001 or ISO 13485 certifications. Supply chain risk management is particularly relevant for cloud services, where reliance on external infrastructure introduces vulnerabilities that must be assessed and mitigated.

#### 2.4.5 AUDITS: THE MOMENT OF TRUTH

This brings us to the topic that instills the most fear among developers new to this field: audits. The word "audit" conjures images of stern inspectors, who scrutinize every document, search for flaws, and issue reports that can derail entire projects. In addition, certainly, an audit can be a stressful experience if you're not prepared. However, it can also be an opportunity for external individuals, with experience and perspective, to help identify areas for improvement that the team, immersed in day-to-day operations, does not see. The attitude with which an audit is approached largely determines what is gained from it.

In the context of ISO 13485, there are two main types of audits: internal and external. Internal audits are conducted by the organization's own staff, who are trained for this purpose, and serve to verify that the quality system is functioning as expected and to identify opportunities for improvement before external audits take place. They are a tool for self-assessment, not for punishment. External audits are conducted by independent bodies, such as certification bodies, and serve to verify that the system complies with the standard's requirements and, where applicable, to obtain the corresponding certification. Both are necessary and complementary.

Preparing for an audit is not something done in the weeks leading up to it but rather the natural result of keeping the quality system up to date throughout the year. If processes are followed correctly, if documentation is up to date, if nonconformities are properly investigated, and if training is current, then the audit will be a relatively straightforward process. The problem arises when the system is neglected and then attempts are made to "fix" it for the audit. Auditors are trained to detect these situations, and when they find them, the mistrust they generate can lead to a much more thorough and demanding audit. The best preparation for an audit is, quite simply, to do things right every day.

During an audit, the auditor will review the system's documentation, interview key personnel,

observe processes in operation, and select project samples to track them in detail. They will look for evidence that the system is actually being implemented, not just on paper. They will ask why certain decisions were made, how specific changes were managed, and what was learned from this or that nonconformity. This is not a hostile interrogation but a systematic investigation to verify that the system fulfills its purpose. Those being audited should respond honestly, show what they have, and if there is something they do not know, acknowledge it without trying to make up answers. Transparency is always the best policy.

When an audit identifies nonconformities, these are documented, and a corrective action plan is needed. It is not the end of the world, far from it. It is simply the acknowledgment that there are aspects of the system that can be improved. The important thing is to respond seriously, investigate the causes, implement the corrections, and demonstrate that the lesson has been learned. Auditors do not expect perfect systems because they know that they do not exist. They expect systems that take continuous improvement seriously, recognize their limitations, and work to overcome them. An organization that demonstrates this attitude usually receives favorable reports, even if it has nonconformities to correct.

For developers facing an audit for the first time, some practical tips can help. The first is not to take the questions personally; the auditor is not judging the person but the system. The second is to answer only what is asked, without adding unsolicited information that might open new lines of inquiry. The third is not afraid to say “I don’t know” if you truly don’t know and to offer to look up the information or refer the auditor to the appropriate person. The fourth is to present records naturally, without excuses or justifications. The fifth, perhaps the most important, is to remember that the purpose of the audit is not to catch anyone out but to verify that the system works and to help improve it.

ISO 13485 certification is not mandatory for all medical software manufacturers, but it is highly recommended and has become a *de facto* requirement in many markets. Obtaining it involves passing an external audit conducted by an accredited body, which verifies that the quality system meets all the standard’s requirements. Once obtained, the certification must be maintained through periodic follow-up and renewal audits. It is a long-term commitment, not a one-time achievement. However, organizations that embrace it discover that the benefits go far beyond regulatory compliance: more organized teams, higher-quality products, fewer errors, greater customer satisfaction, and a culture of continuous improvement that permeates the entire organization.

ISO 13485 is not, therefore, a set of obstacles that regulators impose on developers to make their lives difficult. It is a framework for building organizations capable of producing safe products reliably and sustainably. It is a tool for learning from mistakes, for managing changes carefully, for ensuring that people are trained, for monitoring suppliers, and for conducting honest self-audits. It is, in short, an ally in the task of building software that does not harm those who use it. In a field where the harm can be irreparable, having allies like this is not a luxury but a necessity.

If ISO 13485 teaches us that quality lies in the system that produces the software and not just in the software itself, how do we translate that organizational philosophy into the practice of agile development, where small, autonomous teams value adaptation over planning, without the required discipline turning into rigidity that stifles innovation?

## 2.5 AGILE IN REGULATED ENVIRONMENTS: HYBRID MODELS THAT WORK

“Agile and regulation are not opposites, but complementary.”

There is a profound, almost existential tension between an agile culture and a regulatory culture. Agile manifestos proclaim the value of people over processes, of working software over exhaustive documentation, and of responding to change over the course of a plan. Regulated environments, on the other hand, demand defined processes, rigorous documentation, and strict control over changes. Confronting these two philosophies can create a kind of professional schizophrenia in teams: On the one hand, they know that agile methodologies have proven effective for developing quality software; on the other hand, they sense that the way they apply them in other contexts will not be acceptable when the software could impact people’s health. The result is often paralysis, or worse yet, pretense: agile meetings are held to appear modern, but deep down, work continues in the traditional way because “regulation doesn’t allow for anything else.”

This tension is by no means inevitable. Today, there are hybrid models that integrate the best of both worlds, combining the agility and responsiveness of iterative methodologies with the

discipline and control required by regulated environments. These models are not half-hearted compromises but rather well-thought-out syntheses that respect both the agile philosophy and regulatory requirements. To understand how they work, it is helpful to first debunk some myths that obscure the possibilities for integration. The first is that agile means “no documentation.” The Agile Manifesto states that working software is valued at more than exhaustive documentation, which is very different from saying that documentation has no value. In regulated environments, documentation has unquestionable value, but that does not prevent it from being agile—that is, produced iteratively, just in time, and with the necessary level of detail, not the maximum possible.

The second is that regulated environments require a waterfall approach. For decades, the medical software industry has used sequential models, with clearly separated phases of requirements, design, implementation, and testing. However, this was more due to inertia than to regulatory requirements. Standards such as IEC 62304, which we explore in the next chapter, do not prescribe a specific methodology but rather establish outcomes that must be achieved, regardless of the methodology used. As long as it can be demonstrated that the requirements for traceability, verification, validation, and change control have been met, the method used to achieve this is open. This opens the door to iterative and incremental approaches, provided that they are adapted to generate the necessary evidence.

The third myth is that agility and regulation are incompatible because regulation requires detailed planning, whereas agility prioritizes the response to change. However, this view overlooks the fact that planning in regulated environments does not have to be a straitjacket but rather a framework that allows for managing change in a controlled manner. Agile teams also plan; they just do so in short cycles and continuously adjust. The difference lies in the level of formality and the need to document not only the outcome of planning but also the decisions and changes. An agile team working in a regulated environment can plan its sprints, adjust the backlog, and respond to new requirements, provided it properly documents the process and maintains traceability between what was planned, implemented, and verified.

### 2.5.2 PROVEN HYBRID MODELS: REGULATED SCRUM AND ADAPTED RUP

Several hybrid models have proven effective in practice. One of the best known combines Scrum with quality management practices typical of regulated environments. In this model, work is organized into sprints of fixed duration, with daily coordination meetings, a review at the end of each cycle, and a retrospective for continuous improvement. However, unlike conventional Scrum, each sprint generates not only working code but also the documentation associated with that increment: detailed requirements, test cases, risk analyses, and traceability records. The result of the backlog is not just a list of features but also a managed repository of requirements with all associated information. Every increment considered potentially releasable must meet not only functional acceptance criteria but also regulatory completeness criteria.

Another widely used hybrid model is one based on the adapted rational unified process, which organizes development into phases (initiation, elaboration, construction, and transition) but allows for iterations within each phase. This approach provides the milestone structure that regulators typically appreciate while maintaining iterative flexibility. It is especially useful in large projects or those with distributed teams, where coordination requires a certain degree of formality. The key is that each iteration, within each phase, produces a potentially releasable increment, with all the necessary evidence to demonstrate its quality and safety. In this way, at the end of each phase, there is a partial but fully documented and verified product.

A common element in all successful hybrid models is the early and continuous integration of quality and assurance activities. In traditional models, quality used to be a phase at the end of development, when it was already too late to correct deep-seated problems. In conventional agile models, quality is integrated through practices such as test-driven development, but systematic documentation is often neglected. In hybrid models for regulated environments, quality activities are distributed throughout the entire development cycle, but the necessary records are also generated at the time each activity is performed. This requires teams to incorporate into their definition of “done” not only functional criteria but also documentation and compliance criteria.

### 2.5.3 REQUIREMENTS AND RISK MANAGEMENT IN ITERATIVE CONTEXTS

Requirement management is among the areas in which agile-regulatory integration requires the most attention. In a conventional agile project, requirements are kept in a backlog, often with little detail, and are refined just before implementation. In a regulated environment, requirements must be traceable, verifiable, and linked to risk analyses. This does not preclude progressive refinement, but it requires that each level of detail be documented and that traceability be maintained between high-level requirements and implementation details. Tools such as Jira, when properly configured and equipped with the right plugins, can support this workflow if used in a disciplined manner. The key is that every requirement, at every level of detail, has a unique identifier, is linked to its parents and children, and can be traced back to the code and tests.

Risk management is another area where integration requires a specific approach. In traditional models, risk analysis was typically performed at the start of the project and rarely updated. In an agile approach, risk analysis must be continuous: every new requirement, every design change, and every new feature must be evaluated from a risk perspective. This does not mean conducting a formal analysis every time a line of code is touched but rather keeping the risk register active and updating it as the product evolves. In successful hybrid models, risk management is just another sprint activity, reviewed in planning meetings and updated on the basis of test results and user feedback.

Verification and validation must also be adapted. In an agile project, testing is automated as much as possible and runs continuously. In a regulated environment, in addition to automated testing, formal testing is needed, with documented test cases, recorded results, and retained evidence. This does not preclude automation; in contrast, automation facilitates the systematic generation of records. However, it requires that test cases be linked to requirements, that results be retained in a manner that allows for auditing, and that traceability be maintained between requirements, risks, test cases, and results. Teams that master these hybrid models have continuous integration pipelines that not only run tests but also automatically generate the verification documentation associated with each construct.

### 2.5.4 VERSIONS, RELEASES, AND THE CADENCE OF REGULATED DEPLOYMENT

One of the most interesting challenges of hybrid models is the management of versions and releases. In a conventional agile project, continuous releases are possible, with dozens of deployments per day. In a regulated environment, every release that goes into production must be carefully controlled, documented, and, in many cases, reported to regulators. This does not mean that frequent releases cannot be made, but it does mean that each release must meet the requirements for completeness and evidence. Hybrid models typically establish different release levels: internal releases for testing, validation releases for controlled clinical environments, and production releases for general use. Each level has its own entry and exit criteria, as well as its own documentation requirements. The key is that the flow between levels is seamless and that the evidence generated at lower levels serves to meet the requirements of higher levels.

The branching strategy in code repositories must also be adapted. While in pure agile development, feature branches can be ephemeral and frequent, in regulated environments, each branch contributing to a release must be documented, linked to requirements, and subjected to formal review before merging. Some teams use release branches that are strictly protected, where only changes approved through the formal change control process are allowed. Other practices, such as “force push,” which rewrite the repository history, must be prohibited in these branches to preserve traceability. The discipline in version control is not antitechnical; it is technology applied consciously to ensure reproducibility.

Documentation is, as we have seen, one of the areas where the most friction arises between agile and regulatory cultures. However, it is also one of the areas where the most mutual lessons can be learned. Agile culture reminds us that documentation must be useful, not an end in itself. Regulatory culture reminds us that without documentation, there is no evidence, and without evidence, there is no safety. The key is to document what is necessary, in the most efficient way possible, and at the right time. This means, for example, using wikis instead of static documents, keeping documentation close to the code, automating the generation of reports from data in development tools, and periodically reviewing the documentation to ensure that it remains relevant and up-to-date.

## 2.5.5 CULTURE, TRAINING, AND RELATIONSHIPS WITH REGULATORS

Team training is another critical factor for the success of hybrid models. We cannot expect developers trained in the purest agile culture to simply adapt to regulatory demands, nor can we expect professionals accustomed to regulated environments to accept agile practices without reservation. A conscious effort at training and support is needed to help some understand why documentation is necessary, and others see the benefits of iteration and continuous feedback. The teams that perform best in these environments are those that have developed a shared culture where both perspectives are respected and integrated. This takes time and requires leadership, but it is the only way to build truly effective teams in this demanding field.

The relationship with regulators is also evolving in hybrid models. Regulatory agencies are increasingly open to agile approaches, provided that control and traceability are maintained. In some countries, there are pilot programs for the agile certification of medical software, and some regulators have published specific guidelines on how to apply iterative methods in regulated environments. The key lies in transparency and communication: if a team plans to use a hybrid approach, it must clearly explain this to the regulator, show how it will maintain traceability and control, and demonstrate with facts that the approach works. Regulators are not enemies of innovation, but they need to trust that the processes ensure safety.

For teams just starting out on this path, there are some practical recommendations that can ease the transition. The first is to start with a pilot—a small but meaningful project—where you can test the hybrid model and learn from experience. The second goal is to invest in tools that integrate development management with quality management so that traceability is as automated as possible. The third goal is to establish from the outset the definition of “done,” including regulatory criteria, not just functional ones. The fourth goal is to conduct periodic internal audits to verify that the model is being applied correctly and to identify areas for improvement. The fifth, perhaps the most important, is to cultivate a culture of continuous improvement, where every problem is an opportunity to learn and the system evolves with experience.

The tension between agile and regulated does not disappear entirely in hybrid models but rather transforms into a creative tension that drives continuous improvement. Teams learn to balance the need to respond quickly to changes with the requirement to maintain control and traceability. They learn to document just enough—no more, no less. They learn to plan with enough detail to meet requirements but with enough flexibility to adapt to the unexpected. In short, they learn to navigate the complexity of building secure software without sacrificing the agility that enables innovation. In addition, that ability to navigate complexity is, perhaps, the most valuable skill that teams working at the intersection of technology and medicine can develop.

Ultimately, what hybrid models teach us is that agility and regulation are not opposites but rather complementary. Agility provides the ability to adapt, to learn quickly, and to deliver value continuously. Regulation provides discipline, a systematic approach, and the assurance that what is delivered is safe. Together, they enable the development of medical software that is not only innovative but also reliable and that not only meets users' needs but also does so without putting them at risk. In a world where technology is advancing ever faster and where medicine increasingly relies on it, this synthesis is not an option but a necessity. In addition, the teams that master it will be the ones that truly make a difference in people's health.

Now that we have established the general quality framework with ISO 13485 and have seen how to adapt agility to regulated environments, what specific standard tells us how the medical software lifecycle should be organized to meet safety and efficacy requirements?

## REFERENCES

- Vargesson N. Thalidomide--Birth Defects Res [Internet]. June 1, 2015 [cited May 13, 2026];105(2):140. Available from: <https://pmc.ncbi.nlm.nih.gov/articles/PMC4737249/>
- Master Medical Device Traceability: A Guide for Engineers - Jama Software [Internet]. [cited May 13, 2026]. Available from: <https://www.jamasoftware.com/blog/2025/10/07/how-to-master-traceability-in-medical-device-development/>
- ISO 13485:2016 - Medical devices — Quality management systems — Requirements for regulatory purposes [Internet]. [Cited May 13, 2026]. Available from: <https://www.iso.org/standard/59752.html>