Enhancing the Auditability of the Agile XP Software Development Process in the Context of EU Medical Device Regulations

Thesis submitted in accordance with the requirements of the University of Liverpool for the degree of Doctor in Philosophy by

Mahmood Alsaadi

October 2021

Abstract

Nowadays, there is increasing reliance on software in the healthcare industry, such as software used for diagnostic or therapeutic purposes and software embedded in a medical device, often known as medical device software. Regulatory compliance has become increasingly visible in healthcare industries. Software development companies that develop medical devices software in Europe must comply with EU Medical Device Regulation (EU MDR) regulations in order to get the CE marking.

Agile development practices are increasingly adopted by generic software development companies. For example, agile extreme programming (XP) is now considered a common model of choice for many business-critical projects. The reason behind that is that Agile XP has several benefits, such as developing high-quality software with a low cost and in a short period of time, with the capability to embrace any changing requirements during the development process. However, healthcare industries still have a low rate of agile adoption. This is due to the challenges that software developers face when using Agile XP within the stringent requirements of healthcare regulations. These challenges are the lack of fixed up-front planning, lack of documentation, traceability issues, and formality issues.

Agile software companies must provide evidence of EU MDR conformity, and they need to develop their own procedures, tools, and methodologies to do so. As yet, there is no consensus on how to audit the Agile XP software companies to ensure that their software processes have been designed and implemented in conformity with EU MDR requirements.

The motivation of this research is to assist the companies developing medical device software that wish to adopt Agile XP practices in their effort to meet the EU MDR certification requirements (CE marking). In addition, this research aims to help the information system auditors to extract auditing evidence that demonstrates conformity to the EU MDR requirements that must be met by Agile XP software organisations. This research will try to answer three main questions: Do Agile XP practices support the EU MDR requirements? Is it possible to adopt Agile XP practices when developing medical devices software? Is it possible to submit conformity evidence to EU MDR auditors?

The main aim of this research is to enhance the auditability of the Agile XP software development process in the context of EU MDRs. This aim can be achieved by two main objectives: first, proposing an extension to the Agile XP user story to enhance the early planning activities of Agile XP according to EU MDR requirements. Second, designing an auditing model that covers the requirements of EU MDR. This auditing model should provide the EU MDR auditors with auditing evidence that the medical device software developed with an Agile XP process has fulfilled the requirements of EU MDR.

The main contribution of this research study is the auditing model for EU MDR requirements

that is aligned with the principles of Agile XP. The proposed auditing model would help auditors to audit the Agile XP development process of the medical device with regard to the EU MDR requirements in way of obtaining evidence in conformity to EU MDR requirements. And also, this auditing model can be considered as a guideline that would guide the Agile XP developers to follow the EU MDR requirements. The proposed auditing model has been assessed based on relevant case studies. As result, the evidence gathered shows at least partial support for the requirements in each case study. However, no case study has been demonstrated as supporting fully the auditing yardsticks of the proposed auditing model.

Acknowledgements

Praise is to Allah by whose grace good deeds are completed.

A number of people deserve thanks for their support to bring this research project to fruition. It is therefore my greatest pleasure to express my gratitude to them all.

First and foremost, I must express extend my deepest gratitude to my supervisor, Dr Alexei Lisitsa for his continuous support during my PhD study and related research, for his patience, motivation, and immense knowledge. He guided me throughout my research and the writing of this thesis. I could not have imagined having a better supervisor and mentor for my PhD study. Thank you so much, Dr Alexei Lisitsa, for your dedication and friendly supervision during the last years.

My family, my parents and my brothers provided me with unfailing support and continuous encouragement throughout my years of study and through the process of researching and writing this thesis. This accomplishment would not have been possible without them.

Finally, a special thanks to my wife, Yuser, who has stood by me through all my study journey and encouraged me to pursue my ambitions. She gave me support and help, discussed ideas and prevented several wrong turns. She also supported the family during much of my PhD study. Along with her, I want to acknowledge my three daughters, Habeeba, Hala, and Hareer. They have never known their dad as anything but a student, it seems. Good girls, both, and great sources of love and relief from scholarly endeavour.

To everyone, whether mentioned above or not: thank you.

This thesis is only the beginning of my journey.
Mahmood Alsaadi

Contents

List of Figures

List of Tables

Acronyms

ADP Ambiguities Detection Process.

Agile XP Agile extreme Programming.

ASM Abstract State Machine.

EU MDR EU Medical Device Regulations.

FDD Feature-Driven Development.

IPO Input- Process- Output.

QMS Quality Management System.

SDLC Software Development Life Cycle.

SPICE Software Process Improvement and Capability Determination.

SQuaRE Systems and software Quality Requirements and Evaluation.

UCD Use Case Diagram.

UML Unified Modelling Language.

Chapter 1

Introduction

Regulatory compliance has become increasingly visible in healthcare industries. Healthcare regulations such as the FDA and EU MDR are aimed at improving the quality of health care devices and ensuring health care information privacy and security. Software development companies that develop medical device software must comply with one of these regulations depending on the marketing region. To comply means to act in conformity with what is mandated by the law.

Agile development practices are increasingly adopted by generic software organisations[1]. For example, agile extreme programming (XP) is now considered a common model for choice for many business-critical projects[2]. This is due to the benefits that agile practices provide, such as accelerating the software delivery, low-cost, high-quality products, and enhancing the ability to manage changing priorities. Although agile practices would improve the software development process of the medical device software, there is a low adoption rate of agile practices in the healthcare industry. Software organisations that develop critical systems such as medical devices are usually constrained by regulations. Therefore, developers of medical device software often have concerns about adopting agile practices and this is due to the numerous challenges developers faced when attempting to comply with healthcare regulations such as EU MDR.

Due to the nature of agile software development and the strict requirements of healthcare regulations, several gaps can be uncovered when attempting to manage the development and deployment of software that conforms to EU MDR healthcare regulations. A review conducted by the author of this research has led to the identification of several gaps. These gaps are:

- Up-front planning: agile practices lack the fixed up-front planning as the agile planning approach focuses on delivering the features of the product rather than project activities.
- Documentation: one of the main values of agile practices is working software over comprehensive documentation, which means documents should be as light as possible and discuss only the highest level structures of the software.

- Traceability: in agile practices it is difficult to trace requirements between planning and testing because of the changing of the requirements during the development process.
- Formality: in agile practices requirements and agreements between developers and customers are done in an informal way, such as face-to-face communication and the use of natural language in writing the requirements. Therefore, agile practices lacks precision.

1.1 Related Work

There is currently little publicly available information that suggests there is ongoing research of agile practices within the medical device software domain. The focus of previous studies fall into three categories:

- **Investigation studies**
  Several studies have focused on the usage of agile practices within the healthcare industry. These studies identified the challenges that agile developers faced when developing medical device software. Also, these studies investigated whether agile practices have been adapted in medical device software development and if so, how they were adopted and with what success. For example, [3] conducted a study on agile practices to identify the barriers to following agile practices when developing medical device software. They conducted a survey study with medical device software developers in Ireland to determine the actual barriers that prevented medical software developers from adopting agile practices. They found that barriers to agile adoption included lack of documentation, which is the main barrier to agile adoption, lack of up-front planning, and insufficient coverage of risk management activities. These results were consistent with the findings from our literature review. Another study was conducted by [4] to study the integration of agile practices when developing medical device software. They found that medical device software development organisations can benefit from adopting agile practices. But, the strict requirements of healthcare regulations may prevent these organisations from wholly embracing a single agile practice when developing medical device software. Therefore, they first identified the challenges that agile developers faced when developing medical device software and then they came up with a mixed approach of SDLC to resolve the shortcomings in both agile and plan-driven methods.

  The authors of [5] investigated the suitability of using agile practices in developing medical embedded. They outlined the challenges, such as lack of documentation, traceability issues, and lack of up-front planning and proposed a tailored approach incorporating practices drawn from Agile XP that deals with the challenges related to multiple stakeholder input. They are still working on the other challenges.

The authors of [6, 7] conducted a systematic literature review and systematic mapping study to investigate the suitability of agile methods in the embedded systems development domain. They found that agile methods can be used in developing embedded systems and can bring advantages to embedded system development; however, agile practices need to be modified to suit the more constrained field of embedded product development.

- **Novel methodologies**

  Some studies have proposed a new method for agile practices to be used within medical device software. For example, [8] created TXM (The neXt Methodology) based on agile principles such as adaptive planning, flexibility, iterative and incremental approach in order to make the development of medical devices easier. They achieved TXM using a combination of Agile XP and Scrum. The proposed agile methodology consists of five phases: exploration, planning, development, release, and maintenance. As a result, when XP, Scrum and agile patterns are combined. They cover many areas of the system development life cycle. However, they found that the combination of XP and Scrum can be used directly to develop medical device software.

  Other researchers tried to modify the principles of agile practices to suit the development process within the healthcare industry that is bounded by regulations. The authors of [9] modified the principles of agile practices and came up with their own principles. They developed and piloted a medical device software process assessment framework called MDevSPICE. Moreover, [10] they enhanced the four values and 12 principles of agile practices to make it more suitable to be used when developing embedded systems such as medical devices.

- **Tailoring studies**

  Studies were conducted on tailoring different agile practices or agile practices with other SDLC such as traditional methods and the V-model. For instance, some studies show that combining agile practices could lead to an improvement in the development life cycle of medical device software. Özcan and McCaffery reported that Scrum is a good complement to XP for planning and assessment practices in the development of medical device software[11]. A mixture of XP and Scrum was implemented by a company that develops medical devices in Medtronic [12]. Spence found that the Agile XP practices of pair programming and test-driven development provided early feedback and better quality[12].

  The research reported in [13] implemented a hybrid model of agile methods called AV-Model that is used by a medical device software development company based in Ireland. This agile method consists of tailoring agile practices with plan-driven SDLC, which is the V-model. The purpose of their method is to tackle the issue of changes in requirements at

any stage during development. They found that when the organisation implemented the AV-Model it succeeded in becoming more agile, a number of people in the organisation became more familiar and comfortable with agile practices, the developers were able to make changes in the requirements at any stage of the development process, and there was less chaos in the organisation. After the assessment of the model was completed they found that using this method brought benefits to the organisation, such as a reduction in the project cost and in the effort of rework when change is required. Finally, the authors of [14] found that agile and lean methods can be tailored for regulated development by integrating them with plan-driven practices.

The majority of the previous studies fit in with investigating the use of agile practices within the healthcare industry as well as modifying and tailoring agile practices to suit the healthcare domain in general. However, there is no study conducted on Agile XP practices with particular healthcare regulations such as EU MDR regulations. In addition, it can be seen that there has been no study conducted to provide an enhancement for the auditability of Agile XP software development processes in the environment of healthcare regulations such as EU MDR regulations. Based on the literature review and to the best of the knowledge of the author, no work has been done previously on enhancing Agile XP within the MDR healthcare regulations.

1.2 Research Motivation

The motivation of this research is to increase the adoption rate of Agile XP practices within the medical devices software development that surrounded by regulatory. This can be achieved by assisting the companies developing medical device software that wish to adopt Agile XP practices in meeting the EU MDR certification requirements (CE marking). By providing guidelines that would guide the Agile XP developers to follow the EU MDR requirements. This guidelines are represented by the proposed auditing model for EU MDR requirements that is aligned with the principles of Agile XP. In addition, this research aims to help the information system auditors to easily extract the auditing evidence that demonstrates the conformance to EU MDR requirements of software organisations with Agile XP software processes.

1.3 Research Aim and Objectives

Medical devices software usually developed using the traditional development process such as V-model and Waterfall model. As these models are suitable to be used within regulatory environments where developing phases are rigid and documented. However, these models would make

the development process longer and costly. Alternatively, using agile practices would provide a high quality products with low cost in short time of period.

The main aim of this research is to enhance the auditability of the Agile XP software development process in the context of EU MDRs. This would allow medical device software organisations that adopted Agile XP practices to demonstrate conformity with EU MDR requirements, and also to provide EU MDR auditors with enough evidence that certain steps and activities in developing medical software have been performed in compliance with EU MDR requirements. Moreover, this research aims at answering the relevant questions such as are agile practices suitable to be used in the healthcare industry? Is it possible to adopt Agile XP practices when developing medical device software? And do agile practices support EU MDR requirements?

To achieve the research aim, the following specific research objectives must be achieved:

- Identify the EU MDR requirements that have direct or indirect impacts on SDLC. And conducted an analysis on the extracted requirements of the EU MDR to detect the ambiguities.
- Evaluate the suitability of using agile practices within the healthcare industry. By conducting an analysis on the selected agile practices from the healthcare industry point of view, and based on the identified challenges (up-front planning, documentation, traceability, and formality).
- Identify the gaps between agile XP and EU MDR requirements, by highlighting the main strengths and weaknesses of Agile XP in handling the EU MDR requirements.
- Propose an extension to the Agile XP user story to enhance the early planning activities of Agile XP according to EU MDR requirements.
- Design an auditing model that covers the requirements of EU MDR. This auditing model should provide the EU MDR auditors with auditing evidence that the medical device software developed with an Agile XP process has fulfilled the requirements of EU MDR.
- Present the auditing model more precisely and in an executable manner by applying the ASM model.
- Evaluate the applicability of the auditing model on selected case studies.

1.4 Main Research Contributions

The research contributions of this study are classified into four categories:

1. **Literature on the agile software process in the healthcare industry:**
   Analysis of several studies and surveys related to the agile software process and its implementation in the context of the healthcare industry. This research also analysed the characteristics of the most common agile processes and compared them based on key features for a software development project, such as time, cost, flexibility, and documentation (more details in section 2.2.2). This research also investigates the challenges faced by medical devices software development companies when using an agile process. Moreover, this research has investigated the suitability of the agile practices (XP, Scrum, and FDD) in the healthcare industry, in particular in supporting the EU MDRs. This investigation can help project managers and software engineers select the agile process that best suits the requirements of their software projects (more details in section 2.5).

2. **Extraction of EU MDR requirements**
   The EU MDR has a large number of requirements, but not all of them are related to the software development process that is required for compliance. This research has extracted the EU MDR requirements that have a direct or indirect impact on the SDLC that is needed when applying for a compliance certificate from the EU MDR (more details in section 3.4). As these requirements are written with legal terminology, software developers will find it difficult to understand the requirements. This is due to the ambiguities in the EU MDR requirements. This research has detected the ambiguities in the extracted EU MDR requirements and proposed a solution using software engineering techniques such as UML diagrams to minimise the ambiguities of these requirements (more details in section 3.6).

3. **Extension to the Agile XP user story**
   This research proposes an extension to the Agile XP user story to enhance the conformity to EU MDR requirements for the planning phase. The extension is based on five sub-processes derived from the ASM ground model and SPICE such as ISO/IEC 25030:2019 and ISO/IEC 25010:2011. These sub-processes are: identifying user story suppliers, ensuring the formality of functional requirements, proposing a semi-structured form for non-functional requirements, prioritisation of the user story, and identification of user story dependency (more details in Chapter 4).

4. **An Auditing model for EU MDR requirements in Agile XP environments**
   This study develops an auditing model for EU MDR requirements that is applicable in Agile XP software process environments. The design of the auditing model is based on the framework of evaluation method developed by Scriven (more details in section 5.4). The proposed auditing model consists of four common evaluation components: identification

of the target, design audit criteria and yardsticks, evidence gathering and synthesis, and auditing decision. Moreover, the design of the auditing criteria and yardsticks could not be achieved without the support of quality management system standards such as BS ISO-IEC 12207, BS EN ISO 13485, BS ISO-IEC 25010, and BS EN ISO 14971(more details in section 5.4.2). This auditing model aims to enhance the auditability of Agile XP in regard to EU MDR requirements. The proposed auditing model could help medical device software development companies in their effort to achieve EU MDR certification (CE marking). Also, this auditing model could assist the software auditors of EU MDR in obtaining evidence that demonstrates conformity to the EU MDR requirements as well as helping the software developers to follow the EU MDR requirements.

1.5 Thesis Outline

Chapter 1 presents the definition of the research project, including the research motivation, research aim, the research objectives, the users of the research results, and significance of the research. This chapter also presents the detailed methodology of the research, which is designed to tackle the research objectives. As well as presenting the related work that will focus on previous studies which investigate the adoption and the enhancement of agile practices within medical device software development.

Chapter 2 presents an in-depth literature review of agile software development process in order to give a broader understanding of the main differences and similarities between these agile practices. Also, this chapter presents the investigation result of the adoption rate and the challenges of the selected agile practices in the health care industry. Moreover, this chapter presents a study that conducted to investigate the suitability of the selected agile practices (XP, Scrum, and FDD) in the healthcare industry, in particular in supporting the EU MDRs.

Chapter 3 present an overview of EU Medical Devices Regulations (EU MDR) requirements that that have a direct or indirect impact on the SDLC. Also, this chapter presents an analysis of the ambiguities that were detected in the EU MDR extracted requirements. Moreover, this chapter presents a mapping study that investigates the capability of Agile XP to meet EU MDR requirements.

Chapter 4 provides an extension to the structure of traditional user story of Agile XP, in order to meet the EU MDR requirements for the planning phase. This chapter explain in details the four sub process of the extended XP user story which are identifying user story suppliers, formality of functional requirements, prioritisation of user stories, and identification of user story dependency.

Chapter 5 presents the design methodology of the proposed auditing model for EU MDR requirements to be used in Agile XP environments. This chapter presents a detailed explanation

of the proposed auditing model components which are identification of the target, design audit criteria and yardsticks, evidence gathering and synthesis, and auditing decision.

Chapter 6 explain the evaluation procedure of the proposed auditing model and the auditing results of the selected case studies. In order to investigate the applicability of the proposed auditing model.

Chapter 7 presents a conclusion results of this thesis, as well as its contributions and limitations, and suggestions for future work.

1.6 Related Publication

A number of the outcomes of this thesis have been published (or submitted for publication) in the following journals or conferences:

Alsaadi, M., Lisitsa, A., Khalaf, M., & Qasaimeh, M. (2019). Investigating the Capability of Agile Processes to Support Medical Devices Regulations: The Case of XP, Scrum, and FDD with EU MDR Regulations. *Intelligent Computing Methodologies*, 581-592. `doi:10.1007/978-3-030-26766-7_53.`

Alsaadi, M., Lisitsa, A., & Qasaimeh, M. (2019). Minimising the ambiguities in medical devices regulations based on software requirement engineering techniques. *Proceedings of the Second International Conference on Data Science, E-Learning and Information Systems – DATA '19.* `doi:10.1145/3368691.3368709.`

Alsaadi, M., Lisitsa, A., & Qasaimeh, M. (2021).Extending the Agile XP User Story to Meet EU Medical Devices Regulations. Submitted to *Elsevier Journal of King Saud University – Computer and Information Sciences* (ISSN: 1319-1578).

Alsaadi, M., and Lisitsa, A., (2021). Applying ASM to implement a semi-automated auditing model for EU MDR requirements in Agile XP environments. Submitted to *Journal of Multimodal Technologies and Interaction* (ISSN 2414-4088).

Chapter 2

Literature Review of the Agile Software Development Process

Currently, medical devices rely on software, whether completely such as mobile medical applications, or as embedded software in medical chips such as tele surgery systems. Medical device software is usually developed by using traditional development methods such as the V-model and Waterfall model, as these methods are straightforward and can comply with regulatory requirements such as documentation that ensures traceability. However, these methods could take a long period and could be costly, and also these are inappropriate in cases where requirements change. In contrast, agile practices have several benefits such as developing high-quality software with low cost and in a short period of time with the capability of embracing changes of requirements during the development process. Therefore, companies that develop medical device software can benefit from adopting agile practices. While the adoption rate of agile practices in software development in different industries is increasing, healthcare industries still have a low rate of agile adoption[15]. This is due to the challenges that developers face when using agile practices within the stringent requirements of healthcare regulations.

This chapter will explain the types of Software Development Life Cycle (SDLC) as well as conducting comparisons between them. In addition, this chapter will provide an analysis of several research studies related to the literature review of selected agile software development practices (eXtrem Programming (XP), Scrum, and Feature-Driven Development (FDD)) to give a broader understanding of the main differences and similarities between these agile practices. Moreover, this chapter will investigate the adoption rate and the challenges of the selected agile practices in the health care industry. Finally, this chapter will investigate the suitability of the selected agile practices in the healthcare industry, in particular in supporting the EU MDRs.

2.1 Software Development Life Cycle (SDLC)

The SDLC is a crucial process for achieving high-quality software [16]. The SDLC provides specific guidelines for the software developer on how every step of the life cycle of the software development should be done as well as defining the flow of all activities, actions, and tasks that must be done. There are various SDLC models such as traditional models and agility models; each has advantages and disadvantages and which is chosen is determined based on the project needs. However, the basic phases that should be in each SDLC are: planning, analysis, design, implementation, testing, and maintenance. Each SDLC represents these phases in its own way. The choice of appropriate SDLC model is a significant factor in the success of the type of software project being developed [17].

2.1.1 Traditional SDLC Models

There are several traditional SDLC models, each with its own recognised weaknesses and strengths. Here is a brief description of the most common traditional SDLC models; for a comparative summary of these models see Table 2.1.

- **Waterfall Model**
  This is one of the oldest software developing models, also known as the traditional model. The waterfall model is a sequential process model where each phase must be completed before the next phase begins [18]. It is a rigid step by step process, which means that the developer cannot move from one phase to the next until the phase is completed, and also the developer is unable to go back to the previous phases even when a revision is crucial. The waterfall model usually consists of the basic phases of SDLC, which are planning, analysis, design, implementation, testing, and maintenance. It is frequently applied for critical projects whether small or large, where system requirements have no ambiguity, are well documented, and fixed. The main advantages of this model are that it is simple to use and has a well-defined milestone for each phase. However, this model is not flexible, which means that the cost of this model is high and it needs a long time to complete a project.

- **V-Model**
  The V-model is the modified version of the waterfall also known as (validation and verification) model [19]. The V-model phases can be divided into three categories: verification phases (consists of requirements, analysis and design), validation phases (consists of integration and testing, as well as the maintenance phase), and the implementation phase (the V point where code is built and a prototype system is released).

  This model is the opposite of the waterfall in terms of overlapping, which means that the development phases are inverted after the coding phase is finished. In order to proceed to

the final product, each phase should be checked and approved before moving to the next phase. During the development process of the V-model the software designer and tester are working concomitantly. System test cases are prepared based on the system requirements and on the high-level document (HLD) such as stakeholder needs, whereas implementation test cases are prepared based on the low level document (LLD) such as system features [17, 19]. After the coding phase is completed different tests are applied sequentially as follows: unit testing, integration testing, system testing, and acceptance testing.

Since the V-model is the enhanced version of the waterfall model, both have common advantages and disadvantages. However, what makes the V-model differ from the waterfall model is that the tester is involved from the planning phase, which means that testing begins in the early phases of the software development. In addition, requirements can be changed at any developing phase, but by updating not only the requirements document but also the test documentation. The V-model can be used for small to medium-sized projects where system requirements and specifications are strictly predefined. And also this model can be used when quality is more important than cost or schedule.

- **Iterative Model**
  The iterative model is an incremental approach where a project is divided into small parts called iterations. Each iteration is considered a mini waterfall model that has the basic development phases (planning, analysis, design, implementation, testing, and maintenance) [20]. At the end of each iteration there will be a working version of the software (small release). Each following version will add new functionalities and features to the previous release until all the designed functions of the system are implemented [21, 22]. This would allow the development team to test and debug each release easily as well as releasing the prototype earlier and receive possible feedback from the end user. However, the iterative model is just similar to the waterfall model in terms of flexibility, as it is rigid and no overlapping is allowed, as well as requiring high quality planning and design. The iterative model is usually used in several environments, such as where the requirements of the system are well-defined and complex details are not known, where the system has to reach market early, and when new technology is being tested.

- **Spiral model** The Spiral Model is a software development process that combines the iterative model and the waterfall model, with more focus on risk analysis [16]. The spiral model consists of four main phases, which are:

  - Planning: where the spiral life cycle begins, during this phase the requirements are gathered and a risk analysis is applied.

- Risk analysis: during this phase risks are identified and alternative solutions are provided. At the end of this phase, a prototype is produced.
- Engineering: during this phase, the implementation of the software at different spirals is completed along with the testing of the software.
- Evaluation: where the subsequent spiral is tested by the end user to evaluate if the output functions have met the input requirements. Based on this evaluation, the next spiral iteration is started to implement the feedback suggested by end user.

In the spiral model, the software development process passes through these phases repeatedly in iterations corresponding to various spirals in the model [21, 23]. The move to the next phase is done according to the plan, even if the work on the previous phase is not completed yet. The spiral model is suitable for large and critical projects where risk evolution is vital and documentation is required. However, this model is costly to use and requires highly specific expertise to conduct the risk analysis [24].

Table 2.1: Comparative table for various traditional SDLC models.

Features:	**Waterfall Model**	**V-Model**	**Iterative Model**	**Spiral Model**
Approach	Sequential	Sequential	Incremental	Combination of iterative model and waterfall model
Requirements	Unchangeable	Changeable	Changeable	Changeable
Time	Long	Long	Long	Depends on project size
Cost	High	Low	Low	High
Flexibility	Rigid	Rigid	Rigid (each iteration)	Flexible
User involvement	No	Yes	Yes	Yes
Emphasis	Planning, target dates, documentation, budgets, time schedules and implementation of the project at one time	Validation and verification	Prototyping (release small versions of the system)	Risk analysis and prototype
Formality	Formal	Formal	Semi-formal	Semi-formal
Documentation	Phases are documented	Phases are documented	Phases are documented	Phases are documented
Application area	Large-scale projects such as government projects. When quality is more important than cost or schedule	Large-scale projects such as medical development project. When validation and verification is important	Where system has to reach market early and when new technology is being tested	Large and critical projects, When costs and risk evaluation is important

Despite the fact that traditional SDLC provides a structural development process, these models could be risky and their implantation would lead to failure. This is due to the concerns that developers show when choosing the traditional SDLC. Such concerns are that changing the requirements during the development process is difficult and costly. Another main concern is that the result of the development is invisible for a long time as there is no iteration or prototyping during the development process. This delay can be unsettling to development teams and customers. Therefore, software companies intend to shift to more flexible SDLC such as

agile practices.

2.1.2 Agility SDLC models

The term agile means lightweight, with a quick movement. Martin defined agile software development as "the ability to develop software quickly, in the face of rapidly changing requirements" [25]. Agile models are software development methodologies that contain two sides, a philosophical side and a developmental guidelines side. The philosophical side focuses on customer satisfaction by involving the customer throughout the development process, while the development guidelines focus on software delivery rather than analysis and design processes [26]. Therefore, the agile models are more people-oriented than plan-oriented[27]. In 2001 a group of 17 software developers met together in Utah and proposed a new method under the agile manifesto to alleviate the issues they faced when using the traditional SDLC models. The Agile manifesto consists of four values and 12 principles.

The four values are [28, 29]:

- **Individuals and interactions over processes and tools**: self-confidence over management: people are the most important aspect of project success.
- **Working software over comprehensive documentation**: documents should be short and discuss only the highest level structures in the system.
- **Customer collaboration over contract negotiation**: the involvement of the customer in feedback on a regular and frequent basis leads to a successful project.
- **Responding to change over following a plan**: the response to change at any stage during the development process determines the success or failure of the software project.

The 12 agile principles are [28, 29]:

1. Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
2. Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.
3. Deliver working software frequently, from a couple of weeks to a couple of months, with a preference for the shorter timescale.
4. Business people and developers must work together daily throughout the project.

5. Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
6. The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.
7. Working software is the primary measure of progress.
8. Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
9. Continuous attention to technical excellence and good design enhances agility.
10. Simplicity–the art of maximizing the amount of work not done–is essential.
11. The best architecture, requirements, and designs emerge from self-organised teams.
12. At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behaviour accordingly.

The above values and principles are the solutions that attempt to resolve the difficulty and failure faced by the traditional models that are considered heavyweight and plan-oriented models. The use of agile practices in software development organisations is growing. According to the 14^{th} Annual State of Agile Report, the adoption of agile practices across almost all industries, is rising at an accelerated rate [15]. This noticeable increase in adopting agile models is due to the benefits that the software organisation could obtained from agile models. The 14th Annual State of Agile Report shows the top benefits that encouraged software organisations to adopt agile methodology. These benefits are: accelerated software delivery; where working versions of the system are delivered to the customer in short iterations, enhanced ability to manage changing priorities; increased productivity, enhance software quality, enhanced delivery predictability, reduced project risk, reduced project cost (cost reduction was reported as an important factor for adopting agile practices by software organisations), and increased software maintainability [15, 30].

2.2 Agile processes

In general, the idea behind the agile processes is to do iterative development with the strong involvement of the customer; each iteration contains the basic development phases. At the end of each iteration (one to three weeks), there should be a working version of the software. This approach would guarantee that errors can be discovered in the early phases of development[31]. There are several agile processes such as extreme programming (XP), Scrum, and Feature-Driven

Development (FDD) that have been adopted by software organisations. Although each of these agile processes has a different way of approaching software development, they all share one main feature, which is the close collaboration between the software development team and the customer. This collaboration occurs via face-to-face communication rather than written documentation.

Below is a description of the agile processes that are most widely used in the software development process.

2.2.1 Extreme Programming (XP)

Agile extreme programming, which is known as Agile XP, is one of the most widely used approaches to agile software development. The emphasis of the Agile XP is on coding [29]. Agile XP consist of 12 practices as explained by Kent Beck in his influential book entitled "Extreme Programming Explained: Embrace Change" [29] and explained in [25, 26].

The Agile XP practices are:

The planning game: at the beginning of each iteration, members of the XP team including the customer sit together to collect the system requirements and put them in a card called user story (more details in Chapter 4). After this process is done, they determine the scope of the next release by understanding the business priorities and technical estimations.

Small releases: at the end of each iteration (1-3 weeks) there should be a small release containing the most valuable system requirements.

Metaphor: guide all development process with a simple shared story of how the whole system works. For example, applying words, labels, tags or stories to various elements or parts within the programming processes.

Simple design: XP design usually follows the concept of KIS (Keep It Simple). A simple design is always preferred over a more complex design. A simple design is the one that runs all the tests, has no duplicated logic, and has fewer classes and methods.

Testing: unit tests written by the developer continually to avoid early errors, and the customers write the functional tests to check that requirements are developed as required

Refactoring: developers restructure the system without changing its behaviour to improve the structure of the system. Examples of refactoring are removing duplication, simplifying, or adding flexibility.

Pair programming: all the system code is written by two programmers at the same working

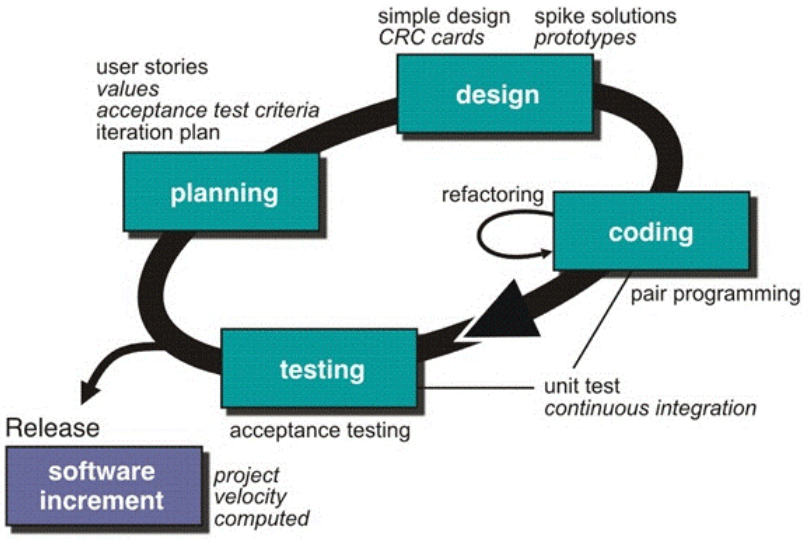

Figure 2.1: The extreme programming process [25].

station. One programmer writes the code and the other checks the code.

Collective ownership: any programmer has the right to change any code anywhere in the system at any time. Programmers share the same responsibilities towards developing the system.

Continuous integration: integration happens several times per day. Every time a task is completed, developers test it and then integrate it into the system.

40-hour week: XP uses the principle of never working overtime, which means the developers can work no more than 40 hours a week.

On-site customer: developers and customers work closely so that they are both aware of each other's issues or errors and they work together to solve those issues.

Coding standards: developers follow certain rules in writing the code for the system. These rules emphasise communication through the code.

These practices would be used throughout each iteration. The iteration consists of four framework activities: planning, design, coding, and testing, as shown in Figure 2.1 and described below:

Planning:
Also known as the planning game, during this phase requirements are gathered from the customer by writing them on user story cards. Then XP team members determine the scope and priority of

the requirements with the estimation of the cost and time for each user story. New requirements can be added to the story card at any time during the development process. The planning game consists of three sub-phases which are: exploration phase, release planning, and iteration planning[25].

Exploration phase: the development team sit together with the customer to explore the system requirements. The customer writes the system requirements on index cards called user story cards. Then the developers work together to estimate these user stories. If one user story could not be estimated, developers could split it into smaller user stories.

Release planning: during this phase the developer will determine the cost, time and value of each user story and will write the estimated values on the user story card. Then the customer will identify the priority of each user story based on business value, and the developers will prioritise the user story based on the risk and velocity. At the end of this phase, the customer and developers will agree on the date of the first release of the system.

Iteration planning: this phase involves further planning of the release planning phase that happens at the beginning of each iteration. In this phase, the user story cards will be converted by the developers into task cards. After tasks are assigned to the programmers, they then will estimate the working hours for each task. Each iteration should be completed within one to three weeks.

Design:

At this phase of the development process, the Agile XP developers must define the main features of the code and assign responsibilities, where each developer will be responsible for the design of a certain part of the code. Agile XP uses the principle of KIS (keep it simple), meaning the design strategy in Agile XP should be as simple as possible [29]. Simple design means the system (code and tests) must be readable, have no duplicate code, have as few classes as possible, and have as few methods as possible [29]. In addition, the design should provide implantation guidance for each user story as it is written, nothing more or less. There are several techniques that Agile XP encourages developers to use to simplify the design, such as a spike solution (operational prototype of part of the design) to have a clear vision of what the system design will look like, refactoring (keeping the design clear), and CRC (class-responsibility-collaboration) cards used to solve the complexity of the design and also to enhance the structure of the design. These techniques provide the development team with guidance on how to improve the design [26]. Moreover, these techniques reduce the cost of changes and allow developers to make design decisions when necessary based on the most current information available.

Coding:

After the planning phase and design phase are completed, the developers move on to writing the unit tests that will check each of the user stories that are to be implemented for the current release. The coding phase is usually achieved by pair programming, which means two programmers working together on one computer to create a code for a user story, with one developer writing the code and the other checking for errors or/and ensuring the code standards. This format provides real-time error solving and real-time quality assurance. Once the code is completed, programmers start to apply the unit test on the implemented code immediately. Then they integrate their code with the works of other pair programmers. This continuous integration that happens daily would help the developer to detect errors in the early stage of the development process [32].

Testing:

Agile XP has four types of tests, which are unit test, integration test, acceptance test, and system test [26, 27]. The unit test is written by the developers and should be in an automated framework, which means it runs automatically. This test occurs on a unit basis that will test every part of the code. The integration test is then performed to check the integration each time a new module is added to ensure that any changes have not affected the functionality of the previously worked module [32]. Once the unit test and integration tests are completed, the developers perform the system tests, which is to validate all the increments as one unit. At the end of each iteration, the customer conducts an acceptance test to ensure that the functionality requirements in the user stories are met.

2.2.2 Scrum

Scrum is an agile software development method that focuses on teamwork activities to produce a quality product in a changing environment [33]. The idea behind Scrum is to have a flexible software development process that only fully defines the planning phase and closure phase [34]. Between these phases software is developed by several teams within a process pattern called sprint. The process of Scrum follows five framework activities: requirements, analysis, design, evaluation, and delivery [26]. Requirements are gathered from the customer during the planning phase, and the development team and customer then create an architectural design for the project. After the initial planning, the product manager identifies tasks and captures them in a list called product backlog, where items can be added to the backlog at any time. The main Scrum activities as explained in [26, 33, 34] and shown in Figure 2.2 are:

Product backlog: This is a list that contains prioritised system requirements, features, and functions that need to be developed within the project. It can be written in the form of user stories or in the form of a to-do list. Items can be added to the product backlog at any time during

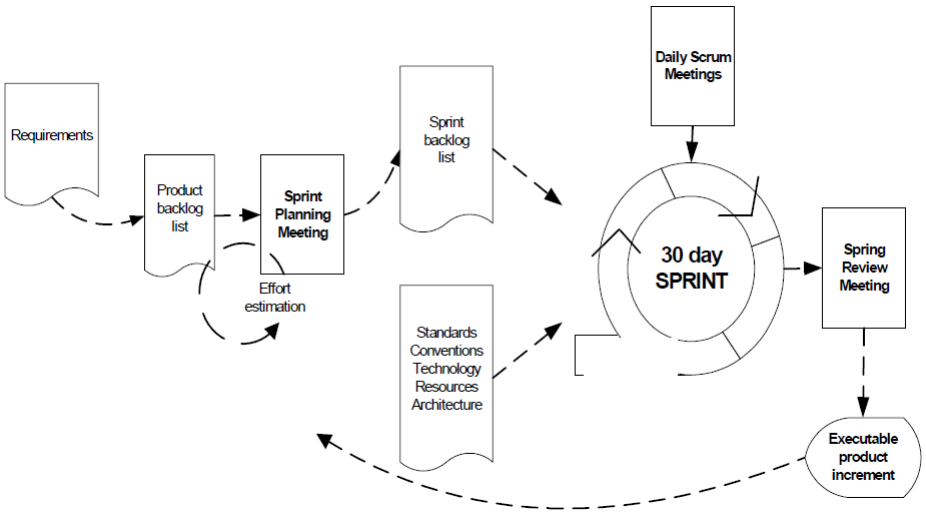

Figure 2.2: Scrum activities [36].

the development process. A product manager can evaluate and update the prioritising of the product backlog list as required. Once the requirements list is fully defined and approved by the product manager, the highest priority tasks will move from the product backlog to the sprint backlog.

Sprint: During sprint, the requirements that are listed in the product backlog will be implemented. Each sprint must fit into a predefined time frame, which is usually 30 days. During the sprint, changes are not welcomed. At the end of the sprint, a review meeting is held to demonstrate the new functions and to get feedback from the customer. The main idea behind each sprint is to deliver a demo to the customer that contains a valuable and functional software increment. The customer can then test the software increment and give feedback.

Daily Scrum meeting: During each sprint, a meeting is held daily by the Scrum team. During this meeting, three questions are asked and answered by the Scrum development team: what did you do since the last team meeting? What are the limitations? And what do you plan to accomplish by the next team meeting? The objectives of this meeting are to address and minimise the risks of the project and to keep everyone informed of the project progress and obstacles.

2.2.3 Feature-Driven Development (FDD)

FDD is an iterative software development process consisting of five phases that mainly focus on the design and building phases [35]. FDD phases as explained in [31, 35, 36] and shown in Figure 2.3 are: *Develop overall model (modelling iteration):* The first phase in FDD is to design

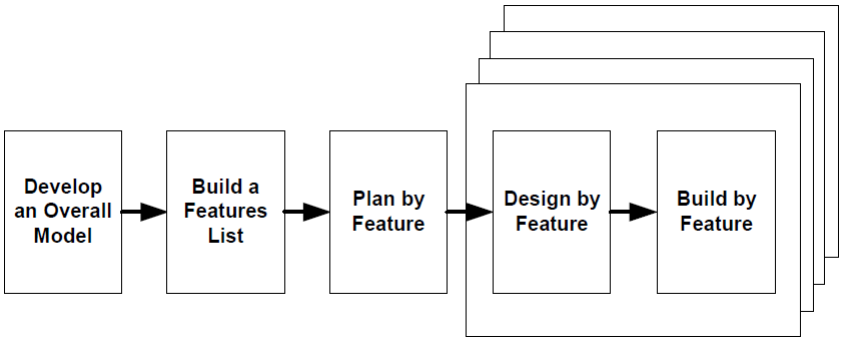

Figure 2.3: Processes of Feature-Driven Development (FDD) [37].

the overall model. The most important part of this phase is gathering the requirements. The user requirements gathering process involves problem domain experts and developers who are familiar with UML modelling and customer needs. The outcome of this phase is the overall system model represented by UML diagrams such as class diagrams that consist of classes, relationships, methods, and attributes. The UML modelling is split into iterations and the developers are divided into modelling groups, and each group creates a UML model that contains features and functions of the system to be implemented. The modelling groups present and discuss the models and then they select the best model. At the end of this phase, the domain experts create a walkthrough document that contains a high-level description of the system.

Build features list: Based on the outcome or the walkthrough documentation of the previous phase, a features list is built. This list can be used later to track the development progress. Features are small pieces of user functional requirements that are written in the form of <action> <result> <object>. If the implementation of the features list takes more than two weeks, it should be split into a smaller features list. At the end of this phase, the features list is reviewed for validation purposes by the customer, domain experts and developers.

Plan by feature: During this phase, a high-level description plan is created as well as prioritising feature sets via an iteration planning meeting. At the end of this phase, the features set is submitted to the programmers as UML classes.

Design by feature: During this phase, developers identify the main classes, methods and properties that are required to build a feature. And also, developers should document this information well in the form of code comments. This phase involves design review meetings, where each developer receives a copy of other developer's design code comments for review purposes, and then the developers give feedback on each other's design.

Build by feature: During this phase, the developer starts to build the feature in an iterative

procedure, in which each iteration should take from two days to two weeks. Building by feature phase involves the following steps: coding, unit testing, integration, and code inspection.

2.3 Agile practices in healthcare industry

Agile development practices are increasingly adopted by generic software organisations [1]. This is due to the benefits that agile practices provide, such as accelerating the software delivery, a low-cost, high-quality product, and enhancing the ability to manage changing priorities. However, software organisations that develop critical systems such as medical devices are usually surrounded by regulatory constraints. Therefore, developers of safety-critical systems often have concerns about adopting agile practices.

This section will explain the adoption rate of agile practices in medical device software organisations as well as the challenges that medical device software organisations face when adopting agile practices.

The Adoption

The use of agile practices in software development organisations is growing. According to the 13th and 14th Annual State of Agile Reports [15, 30], the adoption of agile practices across almost all industries is rising at an accelerated rate, as agile development practices are an attempt to solve the challenges and failures faced when using the traditional SDLC mythologies [37].

Such difficulties are inflexible to change and increase the cost and time of developing software. Adopting agile practices could bring several benefits such as producing a high-quality product with a low cost in a short period of time. According to the 14th Annual State of Agile Reports [15]The top reasons for adopting agile approach by software organisations were stated in are accelerated software delivery, enhanced ability to manage changing priorities, increased productivity, improved business alignment, enhanced delivery predictability, reduced project risk, improved project visibility, improved team morale, reduced project cost, and increased software maintainability.

However, software organisations that develop critical systems such as medical device software must follow certain regulatory constraints. Unfortunately, agile practices and regulated environments are often seen as fundamentally incompatible [38]. Therefore, developers of safety-critical systems often have concerns about adopting agile practices, such as lack of up-front planning, loss of management control, lack of documentation, and regulatory compliance.

Although agile practices would improve the software development process of medical device software, there is a low adoption rate of agile practices in the healthcare industry. According to the reports that are published annually by the State of Agile website [39], the adoption rate

of agile practices in the healthcare industry is between 4% to 7%. Moreover, studies show that there is little data available explaining the experiences of implementing agile practices within medical device software development organisations [1]. Other studies such as [40, 41] and [42] show that some organisations have integrated some of the agile practices into their traditional SDLC and received benefits from incorporating agile practices into their plan-driven approach.

Using agile practices in developing medical devices software would definitely improve the development process and the product quality. Studies by [2, 43] of the adoption of agile practices such as XP in a company that develops medical information systems found that employees thought Agile XP was easy to use, useful and they intended to use it in the future. On the other hand, using agile practices in developing medical devices software would obstruct the regulatory compliance process, and this is the main concern by the medical device organisations for not adopting agile practices [3].

The Challenges

Although agile practices offers many benefits and has been widely used, agile methodologies such as XP do not offer the same benefits when it comes to developing medical device software, because of the challenges of developing within regulatory environments where strict requirements must be followed.

Based on the literature review conducted in this thesis we have identified the challenges that medical device software developers face when using agile practices as their development methodology. There is a gap in the research regarding the applicability of agile practices for safety-critical regulated environments such as medical device software [38]. Studies [3, 5] and [44] show that there are barriers that prevent medical device software organisations from complying with the healthcare regulations if they wish to use agile practices. These challenges can be summarised into four categories:

- **Up-front planning:** Agile practices lack fixed up-front planning, as the agile planning approach focuses on delivering the features of the product rather than project activities.
- **Documentation:** One of the main values of agile practices is working software over comprehensive documentation, which means documents should be as light as possible and discuss only the highest level structures of the software.
- **Traceability:** In agile practices it is difficult to trace requirements between planning and testing, and this is because of the changing of the requirements during the development process.
- **Formality:** In agile practices, requirements and agreements between developers and customers are done in an informal way, such as face-to-face communication and the use of

natural language in writing the requirements. Therefore, agile practices lacks precision.

2.4 Suitability of agile practices in the healthcare industry

In order to investigate the suitability of each agile practice explained previously (XP, Scrum, and FDD) in developing medical device software, we have conducted an analysis on these practices from the healthcare industry point of view, and based on the challenges described in the previous section (up-front planning, documentation, traceability, and formality). A summary of the suitability of agile practices in the healthcare industry is shown in Table 2.2.

2.4.1 Suitability of Agile XP

To check the suitability of Agile XP in the healthcare industry, an analysis have been conducted on what has been used in XP to overcome the challenges that developers faced when dealing with healthcare regulations and Agile XP practice. The following have been found:

Up-front planning: In Agile XP there is no fixed up-front planning as the Agile XP planning approach focuses on delivering the features of the product rather than project activities. In addition, a customer can test each iteration and add a new story at any time of the development process. The Agile XP planning phase is divided into three phases: release planning, iteration planning, daily planning. Therefore, Agile XP lacks fixed up-front planning.

Documentation: XP is focused mainly on programming, and the main concept behind Agile XP is to have less documentation. Documentation in the Agile XP exists in some practices such as user story card, which is considered to be the only document Agile XP has [29] (the user story card is explained in detail in Chapter 4). The user story is then transformed into an acceptance test, and the customer writes the acceptance test which contains exactly what the story should do. Moreover, during the iteration planning, developers use a card called a task card that is used to convert the stories of the iteration into tasks. It is smaller than the whole story, and sometimes one task will support several stories. Each task card consists of the story number, the software engineer, a task estimate, a task description, the software engineer's notes, and task tracking [29]. Finally, in the design phase, developers sometimes use the CRC card (class responsibility collaborator) which is used to identify and organise the classes that are relevant to the system requirements. It is a standard index card that consists of three sections: name of the class, responsibility (attributes and operations that are relevant to the class), and collaborator (the classes that the responsibilities point to) [26]. To sum up, the documentation that exists in Agile XP consists of the user story card, acceptance test card, task card, and CRC card. These documents explain only the very high-level structure of the system, with no deep details.

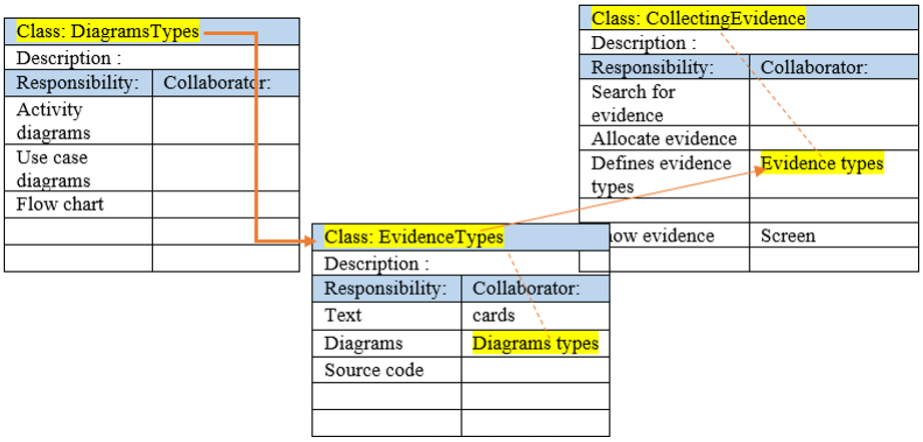

Figure 2.4: Tracing in CRC card.

Traceability: Since the requirements in Agile XP can be changed at any time during the development process, it is difficult to have traceability between requirements, design, code, and test cases. [45] Stated that acceptance tests are not adequate to verify the traceability from end product to customer requirements. However, there are still some XP practices that can be used for traceability purposes, such as the ID of the user story and the acceptance test card that can be used to trace the requirements [46]. Also, in the design phase, the CRC card can be used to trace the requirements by starting with the CRC index card, then going to the collaborator's CRC index card and from there to one of the collaborators and so on, as shown in Figure 2.4.

Formality: The main principle of Agile XP is to deliver a product quickly by focusing mainly on programming and providing prototypes, doing many iterations, and refining the requirements. While the formal methods focus on delivering a product correctly, and any change to the requirements should be recorded in the formal specification [47], in Agile XP, some practices are done informally, for example, requirements are gathered from the customer in an informal way, such as face-to-face communication or via the user story card, and the decisions that are made during the meetings between the developers are not recorded formally.

2.4.2 Suitability of Scrum

To check the suitability of agile Scrum in the healthcare industry, an analysis have been conducted on what has been used in Scrum to overcome the challenges that developers faced when dealing with healthcare regulations and agile Scrum practice. The following have been found:

Up-front planning: In agile Scrum, there is no fixed up-front planning as the Scrum approach focuses on delivering a high-quality product in a changing environment. However, product back-

log can be used as evidence for up-front planning, since it contains everything related to the project such as bugs, customer requirements, competitive product functionality, competitive edge functionality and technology upgrades [48]. During the sprint, the sprint backlog cannot be changed until the sprint has finished, which means that there can be a fixed backlog for the duration of the sprint which is usually 30 days, and this can be considered as semi-fixed planning.

Documentation: Scrum is like other agile practices whose principals have less documentation. So, to the best knowledge of the author and based on the literature review, the documentation in Scrum does exist in the product backlog, development sprint backlog, and model status report which is used for traceability.

Traceability: Scrum uses a model status report for traceability purposes. This report contains key dates, two to five paragraphs of remarks on the project's state, a burn down chart comparing progress to planned work, key metrics (defect inflow, percentage of tests passed, and so on) appropriate to the project's current state, and a list of key risk cycles [49].

Formality: Scrum is a relatively informal process [50]. For instance, deciding what will go into each sprint is done through the daily Scrum meetings by discussing pending problems, prioritising work, and assigning resources to the problems.

2.4.3 Suitability of FDD

To check the suitability of agile FDD in the healthcare industry, an analysis have been conducted on what has been used in FDD to overcome the challenges that developers faced when dealing with healthcare regulations and agile FDD practice. The following have been found:

Up-front planning: in FDD there is up-front planning which occurs in the Develop overall model phase that contains everything related to the project. However, the resulting plan is not necessarily fixed, as requirements can be added, but in a way that should not affect the implemented features. FDD may contain up to 10% change in project requirements without changing the deadline [31].

Documentation: FDD has more documents than other agile practices. In the first phase, FDD produces walkthrough documentation such as UML modelling and a list of features. In the design phase, FDD has design documentation as well as the developer's design code comments. Besides that, FDD has a technical writer, who is one of the developers that creates a description of UML modelling and prepares the user documentation that contains, for example, formal user requirements and decisions [31, 35].

Traceability: The FDD documentation that is mentioned above could be used for traceability

Table 2.2: Summary of the suitability of agile practices in the healthcare industry.

Challenges	XP	Scrum	FDD
Up-front planning	No fixed up-front planning: User story	Semi-Fixed for a specific period: product backlog	Semi-Fixed: Develop overall model phase
Documentation	Light documentation: user story card, acceptance test card, task card, and CRC card	Light documentation: product backlog, development sprint backlog, and model status report	More documentation: UML modelling, list of features, design documentation, user documentation
Traceability	Sometimes, CRC card used for traceability purpose	Model status report can be used for traceability purpose	Features list, UML modelling, user documentation
Formality	Informal	Relatively informal	Relatively informal

purposes, such as features list, UML modelling, user documentation. This documentation can be considered as evidence that makes the FDD to some extent overcome the challenges in traceability between agile practices and healthcare regulations.

Formality: FDD gathers the requirements in a more formal way than other agile practices, for instance, by creating a features list before moving to the next phase, which should be approved by the customer, developer, and domain experts. Also, FDD requires formal documents to be created, such as the UML model and comments, discussion reports, and coding agreement [31]. Since the Agile XP is the main focus of this research, it can be noticed that the Agile XP practice has almost all the challenges that are found between agile practices and the healthcare industry. Therefore, an enhancement should be conducted on Agile XP to make it more suitable for the healthcare industry. Based on the literature studies of the suitability of agile practices in the healthcare industry, the ideal model of agile approach that is suitable to be used in the healthcare industry should be a mixture of different agile practises. This model consists of the basic phases of any SDLC: planning, design, implementation, and testing. These phases are a combination of different agile practices, where each phase is driven by different agile practices that fit the purpose of the phase. Table 2.3 shows the agile ideal model for the healthcare industry.

Table 2.3: Agile Ideal Model for Healthcare Industry.

Phases	**Agile practices**
Planning	FDD: *Develop overall model phase*
Design	Agile XP: *CRC card* or FDD: *UML modelling, code comments*
Implementation	Agile XP: *pair programming* Scrum: *Sprint* FDD*: technical writer*
Testing	Agile XP: *unit test and acceptance test*

2.5 Summary

The use of agile practices in software development organisations is increasing. This is due to advantages such as delivering to the customer high-quality software in a short time at a low cost, and the ability to manage changes in priorities and requirements at any time during the development process that enables the companies to remain flexible and keep up with the fast pace of business. This chapter has given a brief description of what DLC and SDLC types are, with a comparison between them. In addition, this chapter has analysed several research studies related to the literature review of agile software development practices (XP, Scrum, and FDD) to give a broader understanding of the main differences and similarities between the selected agile practices. This chapter has also investigated the adoption rate and the challenges of the selected agile practices in the health care industry. Finally, this chapter has investigated the suitability of each selected agile practice in developing medical device software from the healthcare industry point of view. The following comments can summarise the findings of this chapter:

- There are two types of SDLC, which are traditional models and agile models. The traditional models are the waterfall model, V-model, iterative model, and spiral model. The agile models are XP, Scrum, and FDD.

- The agile models differ from the traditional models. Traditional models are inflexible that follow sequential and a strict order of the development activities, which means developers are not allowed to move to the next development phase until the previous phase is completed. The agile model is flexible and is more people-oriented rather than plan-oriented. The agile models have been proposed to shorten the software development lifecycle with small working deliveries that are fully functional and can be used before the overall project is complete.

- The selected agile practices presented in this chapter include XP, Scrum and FDD. Agile XP mainly focuses on programming, Scrum focuses on teamwork activities to produce a

quality product in a changing environment, and FDD mainly focuses on the design and building phases. Based on the analysed literature in this chapter, it has been found that XP and Scrum are the most deployed and widely used agile software practices.

- Agile practices are increasingly adopted by generic software organisations. This is due to the benefits that agile practices provide, such as delivering high-quality software in a short time and at a low cost. However, there is a low adoption rate of agile practices within the development of medical device software.
- Challenges that medical device software developers face when using agile practices as their development methodology include a lack of the following aspects: up-front planning, lack of documentation, traceability issues, and formality.
- A summary of the suitability of agile practices in the healthcare industry can be found in Table 2.2.

Chapter 3

EU Medical Devices Regulations (EU MDR)

With the significant increase in the use of software in medical devices as a whole system or as an embedded system, medical device software organisations must comply with one of the healthcare regulations based on the marketing region, for example, the Food and Drug Administration (FDA) and Health Insurance Portability and Accountability Act (HIPAA) in the USA and the EU Medical Device Regulation (MDR) in Europe. Since these regulations are written by legislators who are not familiar with software engineering terminology, these regulations can contain ambiguities. As a result of the ambiguities in the regulations, people could have a different interpretation of the legal text. Software developers face challenges in identifying and understanding the regulatory requirements that are related to the software development process.

This chapter focuses on the EU Medical Device Regulations (EU MDR). In this chapter the EU MDR requirements that have a direct or indirect impact on the SDLC will be extracted. Then, an analysis will be conducted on the extracted requirements of the EU MDR to detect the ambiguities. This chapter presents a solution that could minimise the ambiguities of the extracted EU MDR requirements. This solution is based on software engineering techniques such as user story cards and UML diagrams. Finally, this chapter presents a mapping study that investigates the capability of Agile XP to meet EU MDR requirements. Some sections of this chapter have already been published by the author in [51].

3.1 Medical devices regulations

The significant rise of digital healthcare, medical applications, and software in medical devices has led to a greater attention to healthcare regulations [52]. Regulations mean rules or direc-

tives made and maintained by authorities. Healthcare service providers such as medical device software organisations must follow these regulations. Medical device software organisations must obtain approval and comply with the healthcare regulations of the marketing region. Healthcare regulations are enacted to ensure a high level of protection of health for patients and users, as well as setting high standards of quality and safety for medical devices to meet common safety concerns as regards such products [53]. Different regions follow different regulations, for example, in the United States, healthcare services providers must comply with the FDA and Health Insurance Portability and Accountability Act (HIPAA), while in Europe they must comply with Medical Device Regulations (MDR). These are two of the most common healthcare regulations, and are explained in more detail below:

Food and Drug Administration (FDA): FDA is a public health agency that protects American customers by enforcing the Federal Food, Drug, and Cosmetic Act and several related public health laws [54]. The FDA aims to ensure that medicines and medical devices are safe and effective. The FDA has classified medical devices into three classes, I, II, and III, based on their risk level. Each class has specific requirements for compliance purposes. For example, medical devices regulated as Class I are required to have general controls, such as quality system requirements stated by the FDA; Class II requires special controls such as performance standards; and Class III requires up-front pre-market approval [55].

Health Insurance Portability and Accountability Act (HIPAA): HIPAA is a set of rules that aim to protect sensitive health information [56]. Privacy and security are the main focus of HIPAA rules to ensure data protection, which is known as protected health information (PHI). HIPAA has classified organisations into two categories: covered entities (an entity that provides healthcare services and processes data electronically) and business associates (people or organisations that develop software for a covered entity) [55]. Therefore, any organisations in the US dealing with PHI or electronic protected health information (ePHI) must comply with HIPAA regulations. Our main focus in this research is on the EU MDR regulations

3.2 EU Medical Device Regulations (MDR)

Medical Device Directive (MDD) 93/42/EEC and Active Implantable (AIMD) Medical Devices Directive 90/385/EEC were the previous Europe Union healthcare regulations for medical devices. On 5 April 2017, the official journal of the European Commission published the revised framework of regulations which combines the two directives, MDD and AIMD, and was introduced as a Regulation instead of a Directive, called Medical Device Regulations (MDR) [53]. The significant change from previous versions is that stand-alone software and mobile apps are now considered as medical devices and should be developed per healthcare regulations. The aim

of EU MDR as stated by the European Commission is to ensure the smooth functioning of the internal market as regards medical devices, taking as a baseline a high level of protection of health for patients and users, and taking into account the small- and medium-sized enterprises that are active in this sector [53].

EU MDR Definitions

Terms that are related to our research are defined by EU MDR in Article 2 of Chapter 1 of Regulation (EU) 2017/745[53] and by the Medical Device Coordination Group [57] as follow:

Medical devices: *'any instrument, apparatus, appliance,* ***software****, implant, reagent, material or other article intended by the manufacturer to be used, alone or in combination, for human beings for one or more of the following specific medical purposes:*

- diagnosis, prevention, monitoring, prediction, prognosis, treatment or alleviation of disease,
- diagnosis, monitoring, treatment, alleviation of, or compensation for, an injury or disability,
- investigation, replacement or modification of the anatomy or of a physiological or pathological process or state,
- providing information by means of in vitro examination of specimens derived from the human body, including organ, blood and tissue donations,'

Medical device software (MDSW): *'medical device software is software that is intended to be used, alone or in combination, for a purpose as specified in the definition of a "medical device" in the MDR'.*

Active medical device: *'means any device, the operation of which depends on a source of energy other than that generated by the human body for that purpose, or by gravity, and which acts by changing the density of or converting that energy. Devices intended to transmit energy, substances or other elements between an active device and the patient, without any significant change, shall not be deemed to be active devices. Software shall also be deemed to be an active device'.*

System: *'means a combination of products, either packaged together or not, which are intended to be interconnected or combined to achieve a specific medical purpose'.*

User: *'means any healthcare professional or lay person who uses a device'.*

Electronic programmable systems: *'are devices that incorporate electronic programmable systems and software that are devices in themselves'.*

Stand-alone software: *'means software which is not incorporated in a medical device at the time of its placing on the market or its making available'.*

Software: *'set of instructions that processes input data and creates output data'.*

Expert function software: *'software which is able to analyse existing information to generate new specific information according to the intended use of the software'.*

3.2.1 EU MDR medical device classification

If software meets of one the EU MDR definitions explained previously, then the software is regulated as a medical device. The next question is to which class of EU MDR it belongs. In EU MDR, medical device software is assessed based on class type. Therefore, medical device software organisations will need to determine the class type of their systems in order to identify the route to compliance and CE marking certification. EU MDR has classified medical devices in Chapter III of Annex VIII classification rules into four classes based on the risk level of the devices as follows [53]:

Class I: generally classified as a low risk class;

- All other active devices are classified as class I.

Class IIa: generally classified as a medium risk class;

- All active therapeutic devices intended to administer or exchange energy.
- Active devices intended for diagnosis and monitoring.
- Software intended to provide information that is used to make decisions with diagnostic or therapeutic purposes.
- Software intended to monitor physiological processes.
- Devices specifically intended for recording diagnostic images generated by X-ray radiation.

Class IIb: generally classified as a medium risk class;

- All active devices intended directly to control or monitor the performance of active therapeutic. Active therapeutic device is any active device used, whether alone or in combination with other devices, to support, modify, replace or restore biological functions.
- Software intended to provide information which is used to take decisions with diagnosis or therapeutic purposes that have an impact and may cause a serious deterioration of a person's state of health or a surgical intervention.

Class III: generally classified as a high-risk class;

- All active devices that are intended for controlling, monitoring or directly influencing the performance of active implantable devices.
- All active devices that make decisions have an impact and may cause death or an irreversible deterioration of a person's state of health.

3.2.2 EU MDR requirements

EU MDR has several requirements related to medical device software. These requirements are outlined in annexes of legislative acts of medical devices that are published by the *Official Journal of the European Union:*

- Annex I General safety and performance requirements.
- Annex II Technical documentation.
- Annex IV EU declaration of conformity.
- Annex VII Requirements to be met by notified bodies.
- Annex IX Conformity assessment based on a quality management system and assessment of the technical documentation

However, this research focuses on the requirements that directly or indirectly affect the SDLC of medical device software.

3.3 Extracted requirements of EU MDR

It was challenging to distinguish between requirements that are related to the SDLC and requirements that are related to medical device software. However, EU MDR requirements that would have an impact on the development phases of the medical device software have been extracted (see Appendix A). Figure 3.1 shows a summary of the extracted requirements of EU MDR. Most of the extracted requirements have been found in annex I, annex II, and annex IX.

3.4 Compliance with EU MDR

Medical device software development companies in Europe that develop software systems or embedded systems for medical purposes must comply with the EU MDR regulations in order to be placed in the European market [58]. Compliance means conforming to the requirements of

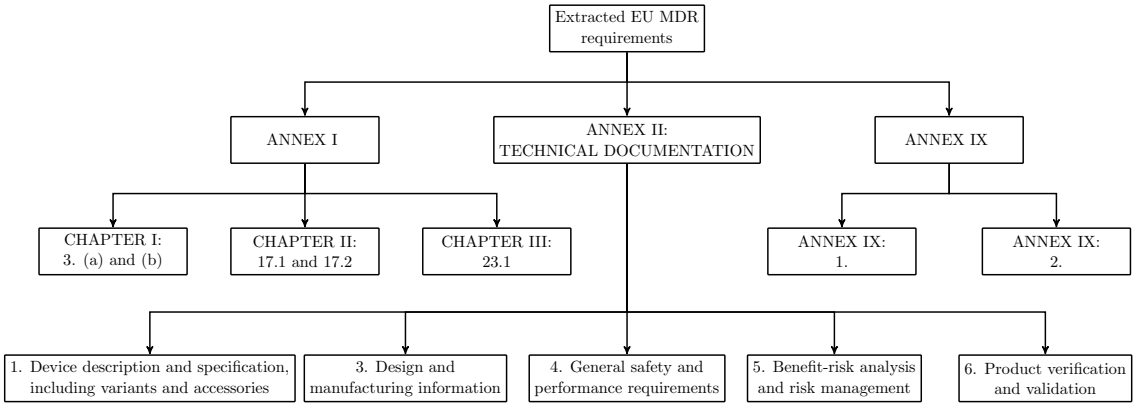

Figure 3.1: Extracted requirements of EU MDR.

EU MDR to get the CE marking [59]. CE marking is the medical device manufacturer's claim that a product meets the General Safety and Performance Requirements (GSPR) of all relevant European Medical Device Regulations (MDR) and is a legal requirement to place a device on the market in the European Union [58]. To obtain this CE marking, audits are performed by notified bodies within the country of the developed systems to ensure their safety and reliability. In addition, medical device software companies must provide technical documentation, a risk analysis, proof of compliance with the essential requirements of EU MDR, and a declaration of conformity issued by the manufacturer [60].

As compliance procedures depend on the classification of the medical device, specific requirements must be provided and an appropriate conformity assessment procedure can be chosen. Different conformity routes mean different lists of requirements should be provided by the manufactures. Table 3.1 shows a description of the EU MDR conformity assessment procedures and the list of requirements to be provided.

3.5 The ambiguities in EU MDR requirements

As mentioned above, medical device software companies in Europe must comply with the EU MDR requirements. Since the EU MDR requirements are enacted by legislators and written in a legal document, these regulations are typically written with ambiguities [61]. Ambiguity occurs when a statement lacks relevant information or when a word or phrase has more than one possible interpretation [62]. Ambiguities are widespread in regulations and laws, due to legal terms that have multiple interpretations [63]. Therefore, people may have a different interpretation and understanding of the regulations.

Software developers developing medical devices software could face challenges in identifying the relevant requirements and understanding the exact meaning of these requirements, for exam-

Table 3.1: EU MDR Conformity Assessment procedures.

Classification	Requirements to be provided	Auditors
Class I	• Annex II: technical documentation. • Annex IX: conformity assessment based on a quality management system and on assessment of technical documentation. • Annex IV: EU declaration of conformity.	Self-assessment
Class IIa	• Annex II: technical documentation. • Annex IX: conformity assessment based on a quality management system and on assessment of technical documentation. • Annex IV: EU declaration of conformity	Notified body assessment
Class IIb	• Annex II: technical documentation. • Annex IX: conformity assessment based on a quality management system and on assessment of technical documentation. • Annexes X and XI: Type examination and production QMS. • Article 52 Para 4 – Assessment of Technical Documentation of a representative device of each generic device group. • Annex IV: EU declaration of conformity	Notified body assessment
Class III	• Annex II: technical documentation. • Annex IX: conformity assessment based on a quality management system and on assessment of technical documentation. • Article 52 Para 4 – Assessment of Technical Documentation of a representative device of each generic device group. • Annex IV: EU declaration of conformity.	Notified body assessment

ple, difficulties in identifying the regulatory requirements that have impacts on the SDLC [64]. Understanding regulatory requirements is an essential aspect of the compliance process [65].

Ambiguous EU MDR requirements could be one of the obstacles to regulatory compliance for medical device software. Failures of compliance with healthcare regulations occur either due to accidental misuse or malicious misuse [66]. Limited understanding of the regulatory requirements is one of the main reasons for accidental misuse [67]. Therefore, there is a need to have precise requirements that contain terms that are relevant to software engineering and can be understood by medical device software developers.

3.5.1 Ambiguity in requirement engineering

Ambiguity from a software engineering point of view occurs if there is more than one interpretation for the requirement, then that requirement is probably ambiguous [68]. Ambiguities in requirements can be classified into two categories: language ambiguities which can be observed only by the reader who knows the language well, and engineering ambiguities which depend on the domain of the software to be implemented and can be observed by the reader who has experience in that domain [69].

Several types of ambiguity mentioned in an ambiguity handbook could occur in requirement engineering, such as [70]:

- Lexical Ambiguity: When a word has multiple meanings.
- Analytical Ambiguity: When the role of the constituents within a phrase or sentence is ambiguous.
- Elliptical Ambiguity: When it is not certain whether or not a sentence contains an ellipsis.
- Vagueness and Generality: When it is not clear how to measure whether the requirement is fulfilled or not.

3.6 Detecting the ambiguities of EU MDR requirements

This section will conduct an analysis of the extracted EU MDR requirements to detect the ambiguities in these requirements. The ambiguities detection process (ADP) was used in detecting the ambiguities of the extracted requirements of EU MDR, as shown in Figure 3.2. Firstly, the keywords of the original text are defined, and then an English dictionary such as the Cambridge Dictionary is used to explore the meaning of each defined keyword. After that, a reasoning of ambiguity is performed to explain why the keywords are ambiguous from a software engineering perspective. Keywords are considered to be ambiguous if they have at least two different valid interpretations. Finally, an ambiguity type of the keywords is determined.

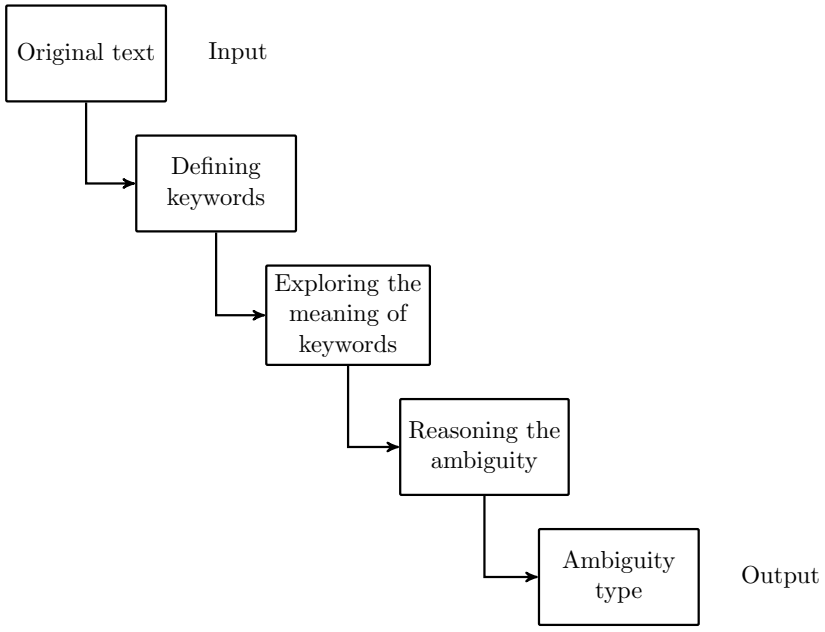

Figure 3.2: Ambiguities Detection Process (ADP).

3.6.1 Examples of detecting the ambiguity

Most of the ambiguities were found in the EU MDR technical documentation. In these examples, we have applied the ambiguities detection process on the extracted requirements of the EU MDR technical documentation that have an impact on the SDLC. Several examples of detecting the ambiguities in EU MDR are shown below:

Example 1:

Original text: *Annex II 1.1. Device description and specification.*

Keywords: description and specification.

Meaning of keywords:
Description: a statement or a piece of writing that tells what something or someone is like.
Specification: a detailed description of how something should be done, made, etc.

Reasoning of ambiguity: In software engineering, there are many types of description forms and specifications. 'Specification' in software engineering has multiple meanings such as a written document, a set of graphical models, a formal mathematical model, and a collection of usage scenarios, a prototype, or any combination of these [26, 71]. Therefore, EU MDR should specify which types of specification and descriptions of the system are required to be in the technical documentation.

Ambiguity type:

- Lexical Ambiguity,
- Elliptical Ambiguity, and
- Vagueness and Generality.

Example 2:

Original text: *Annex II 1.1. a. product or trade name and a general description of the device including its intended purpose and intended users;*

Keywords: general description

Meaning of keywords:
'General description' indicates that they are talking about something as a whole, rather than about a part of it.

Reasoning of ambiguity: it is unclear how to measure the limit of the general description to see whether it fulfils the requirement or not.

Ambiguity type:

- Vagueness and Generality

Example 3:

Original text: *Annex II 1.1. b. Principles of operation of the device and its mode of action, scientifically demonstrated if necessary;*

Keywords: operation, mode, and action.

Meaning of keywords:
'Operation': the fact of operating or being active. 'Mode': formal, a way of operating, living, or behaving.

Reasoning of ambiguity: The word 'operation' here has multiple meanings: the activity of the device, the function of the device, and the performance of the device. So what is meant by the operation here? Also, it is not clear what is meant by 'mode of action'. Is it a type of process, or a type of behaviour?

Ambiguity type:

- Lexical Ambiguity

Example 4:

Original text: *Annex II 1.1.j. A general description of the key functional element, e.g. its parts/components (including software if appropriate), its formulation, its composition, its functionality and, where relevant, its qualitative and quantitative composition. Where appropriate, this shall include labelled pictorial representations (e.g. diagrams, photographs, and drawings), clearly indicating key parts/components, including sufficient explanation to understand the drawings and diagrams;*

Keywords: functional elements

Meaning of keywords:
'Functional elements': parts or components, which work together to control a task or activity.

Reasoning of ambiguity: 'Elements' in software development are divided into three categories: input elements, process elements, and output elements. EU MDR is required to have a general description of the key functional elements. For a software developer, it is not clear what types of elements are required and which diagrams they should represent. Software engineering has multiple diagrams that can represent the functional elements of the software.

Ambiguity type:

- Vagueness and Generality.

Example 5:

Original text: *Annex II 1.1.l. technical specifications, such as features, dimensions and performance attributes, of the device and any variants/configurations and accessories that would typically appear in the product specification made available to the user, for example in brochures, catalogues and similar publications*

Keywords: technical specification

Meaning of keywords:
'Technical specification': this is a document that defines a set of requirements that a product or assembly must meet or exceed.

Reasoning of ambiguity: Software engineering has the term 'requirements specification', which is often used to refer to a document that contains a detailed description of the requirements for a system, including functional requirements, user interface, and design requirements. Therefore, it is not clear here for software developers what kind of technical specification is required and how the details should be.

Ambiguity type:

- Vagueness and Generality.

We have followed the same process as above to detect the ambiguities for the whole of the extracted EU MDR requirements. The majority of the ambiguities types were lexical ambiguity, where keywords have multiple meanings, and vagueness and generality, where it is not clear how to measure whether the requirement is fulfilled or not. Table 3.2 shows a summary of detected ambiguities in EU MDR extracted requirements.

3.7 Minimising the ambiguity in EU MDR requirements

As mentioned previously, for software developers developing a system, compliance with the regulations can be very challenging. Therefore, EU MDR should contain software engineering terms so that it can be easy for the software developer to understand and identify the relevant requirements. This section will present a method to minimise the ambiguities in EU MDR extracted requirements based on software engineering techniques such as user story card and UML diagrams.

3.7.1 Methodology of minimising the ambiguity

The Input- Process- Output (IPO) method was used as the methodology to minimise the ambiguity of EU MDR as shown in Figure 3.3. IPO is an approach that is widely used in software engineering and systems analysis for describing the structure of information processing [72]. The input will be the extracted requirements of EU MDR that have ambiguity. Moving to the next step, which is the process, during this step paraphrasing of the ambiguous keywords is conducted to put them into terms that are more relevant to software engineering and then integrate them into the original text. The original text with the integrated software engineering terms is represented in the form of a user story card. The final step in this process is the output, which is when a user story card is converted into Unified Modelling Language (UML) use case diagrams to illustrate the EU MDR extracted requirements for the software developer in a precise way. This methodology can be used by medical device software auditors and software engineers to have a clear vision of the EU MDR requirements that could help in achieving compliance.

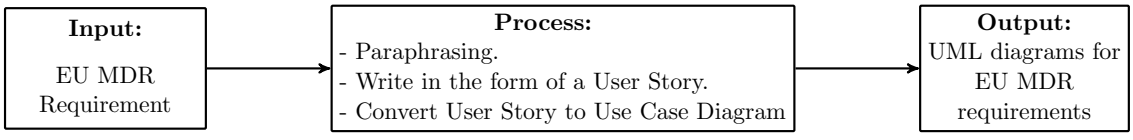

Figure 3.3: IPO process for minimising the ambiguity of EU MDR.

Table 3.2: Summary of Ambiguities in MDR Software Requirements.

EU MDR Extracted Requirements	**Ambiguous Keywords**	**Ambiguity Type**	**Reasoning of Ambiguity**
Annex I, chapter II, 17.1.	Repeatability, reliability and performance	Lexical Ambiguity Vagueness and Generality	Adjectives can make requirements untestable
Annex I, chapter II, 17.2.	principles of development life cycle	Vagueness and Generality	No specific SDLC
Annex I, chapter III, 23.1.	General requirements	Vagueness and Generality	Generality
Annex II, technical documentation, 1.1.	Description, and Specification	Analytical Ambiguity. Elliptical Ambiguity. Vagueness and Generality	'Specification' in software engineering has multiple meanings. Which types of specification and description of the system are required?
Annex II 1.1. *a.*	General description	Vagueness and Generality	It is unclear how to measure the limit of the general description and whether it fulfilled the requirement or not
Annex II 1.1. b.	Operation, Mode, and Action	Lexical Ambiguity	In software engineering, these keywords have multiple meanings
Annex II 1.1.g.	Novel features	Lexical Ambiguity	'Novel features' has two meanings: new or original
Annex II 1.1.h.	Accessories	Elliptical Ambiguity	'Accessories' usually means extra hardware that functions with the software. So it is not clear for software developers what is meant by a description of the accessories.
Annex II 1.1.j.	General description of functional elements	Vagueness and Generality	Elements in software development are divided into three categories: input elements, process elements, and output elements
Annex II 1.1.l.	Technical specification	Vagueness and Generality	It is not clear here for software developers what kind of technical specification is required and how the details should be
Annex II 3. B.	Specifications and Manufacturing	Lexical Ambiguity.	Multiple meaning
Annex II 4.	General safety and Performance requirements	Vagueness and Generality	Generality

This methodology was applied on the extracted requirements of Annex II technical documentation of EU MDR (which contains the majority of the EU MDR requirements that have an impact on the SDLC). The results are shown below:

User story card:

This section illustrates the results of paraphrasing and rewriting in the form of a user story format for each extracted requirement of Annex II technical documentation:

Table 3.3: User story card for Annex II 1.1.

Story ID: TD.3.	**Story Ref:** Annex II technical documentation
User Story: Annex II 3. DESIGN AND MANUFACTURING INFORMATION	
As: < MDR's Auditor> **I want:** < the technical documentation to have information about the software design and implementation stages> **So that:** < meet the requirements of Section 3 of ANNEX II of EU MDR >	
Acceptance criteria	
And I know I am done when: Design and implementation documents have the following: • (a) Information about software design stages, such as database relational tables, UML diagrams. • (b) Complete information about SDLC phases such as planning, design, implementation, testing.	

Table 3.4: User story card for Annex II 3.

Story ID: TD.1.	**Story Ref:** **A**nnex II technical documentation
User Story: Annex II 1.1**.** DEVICE DESCRIPTION AND SPECIFICATION	
As: < MDR's Auditor> **I want:** < the technical documentation to have information about the software description and specification > **So that:** < meet the requirement of Section 1 of ANNEX II of EU MDR >	
Acceptance criteria	
And I know I am done when: device description and specification document have the following: • (b) Product name, general description of the software functionality, purpose of software, and user target. • (d) Functional requirements of the software. • (f) Identification of the risk type of the software. • (g) Explanation of non-functional requirements of the software. • (H)Description of any accessories that are to be used with the software (hardware specification). • (j) For each essential functional requirement: its components, its functionality, its codes, and UML diagrams. • (l) Software technical specifications: a document that has detailed requirements for software, including functional requirements, interface (GUI), and design requirements.	

Table 3.5: User story card for Annex II 4.

Story ID: TD.4	**Story Ref:** Annex II technical documentation
User Story: Annex II 4. GENERAL SAFETY AND PERFORMANCE REQUIREMENTS	
As: < MDR's Auditor> **I want:** < the technical documentation to have detailed information about the software non-functional requirements that are set out in Annex I > **So that:** < meet the requirement of Section 4 of ANNEX II of EU MDR >	
Acceptance criteria	
And I know I am done when: Non-functional requirements documents have the requirements of Annex I as follows: • Annex I – 17.1. To ensure repeatability, reliability and performance in line with their intended use. • Annex I- 17.2. Principles of the development life cycle, risk management, including information security, verification and validation. • Annex I- 17.3. If the software is to be used on a mobile platform, it shall have information about specific features of the mobile platform such as the size and contrast ratio of the screen.	

Table 3.6: User story card for Annex II 5.

Story ID: TD.5.	**Story Ref:** Annex II technical documentation
User Story: Annex II 5. BENEFIT-RISK ANALYSIS AND RISK MANAGEMENT	
As: < MDR's Auditor> **I want:** < the technical documentation to have detailed information about the software risk analysis and risk management > **So that:** < meet the requirement of Section 5 of ANNEX II of EU MDR >	
Acceptance criteria	
And I know I am done when: software risk analysis and risk management document have the following: • (a) Risk management life cycle: (identify, analyse, plan, track, and control). • (b) The solutions adopted and the results of the risk management.	

UML use case diagrams (UCD):

In software engineering, use case diagrams are used to describe the interactions between users and the system [73]. UCD contains two main components, which are use cases and actors. The use cases represent a set of actions that the system provides to actors, where actors are parties

outside the system that interact with the system, which can be humans or other systems [74]. Here, the use cases will represent the EU MDR requirements and the actor will represent the auditor. Using use case diagrams would help the software developers and the auditors to define and organise the EU MDR requirements.

This section illustrates how each user story can be converted into use case diagrams. These use case diagrams shows how the EU MDR's auditor can interact with the required requirements of EU MDR. For example, in Figure 3.5 the EU MDR's auditor will require to have these case studies; (information about software design stages) and (complete information about SDLC phases) each with its ¡¡include¿¿ case studies. This would help the auditor and the software developers to easily identify the required requirements of EU MDR.

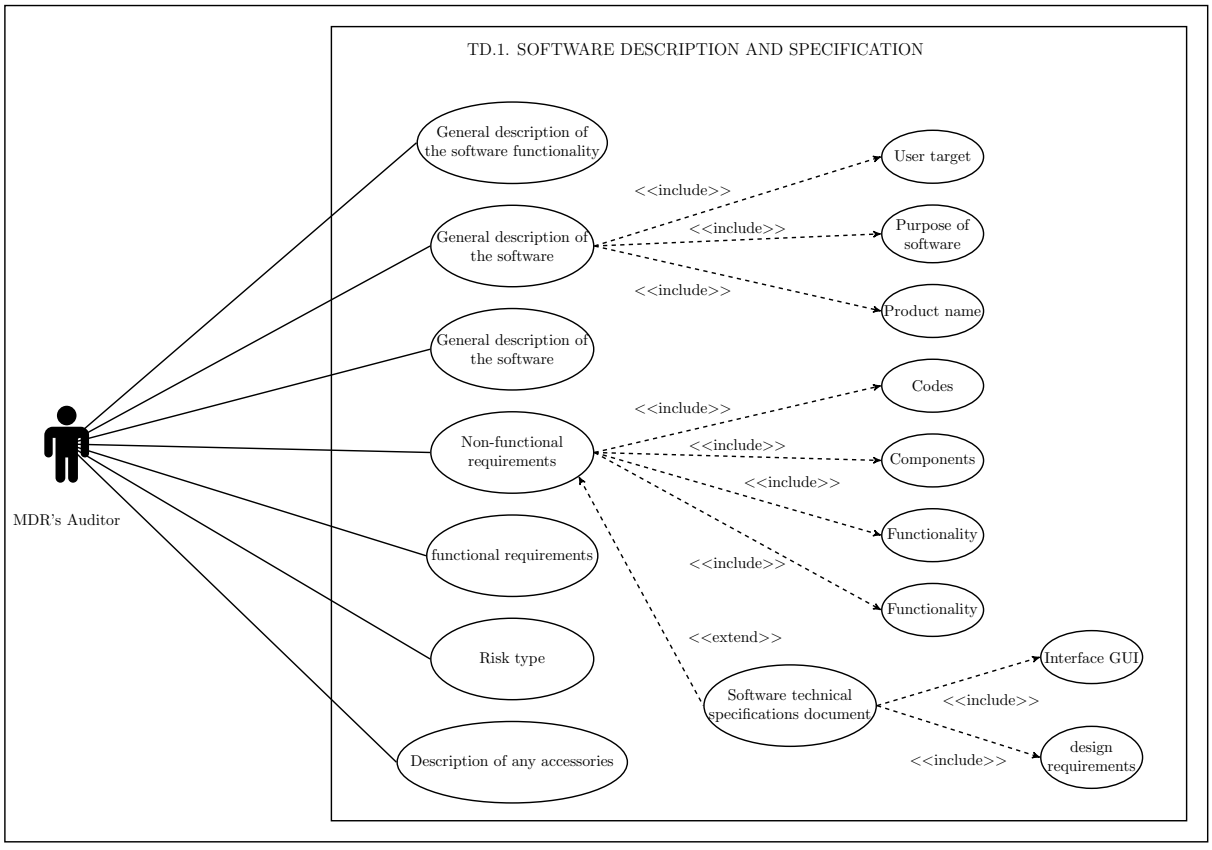

Figure 3.4: Use Case Diagram for Annex II 3.

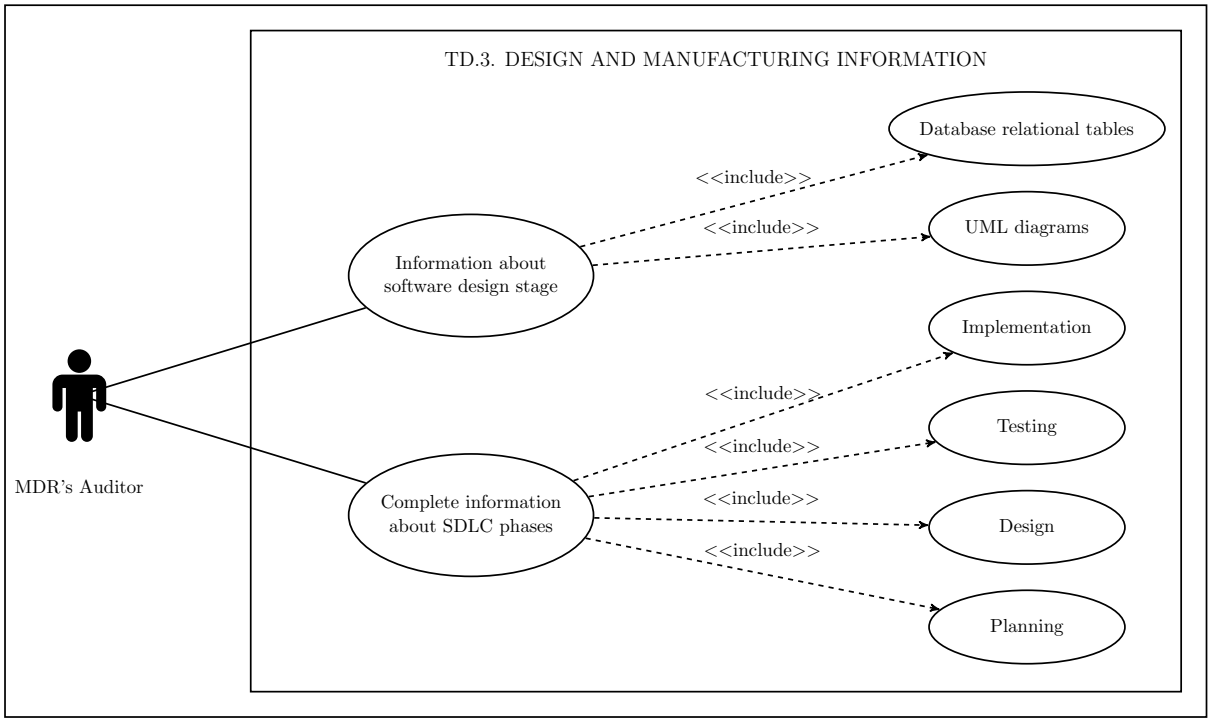

Figure 3.5: Use Case Diagram for Annex II 1.1.

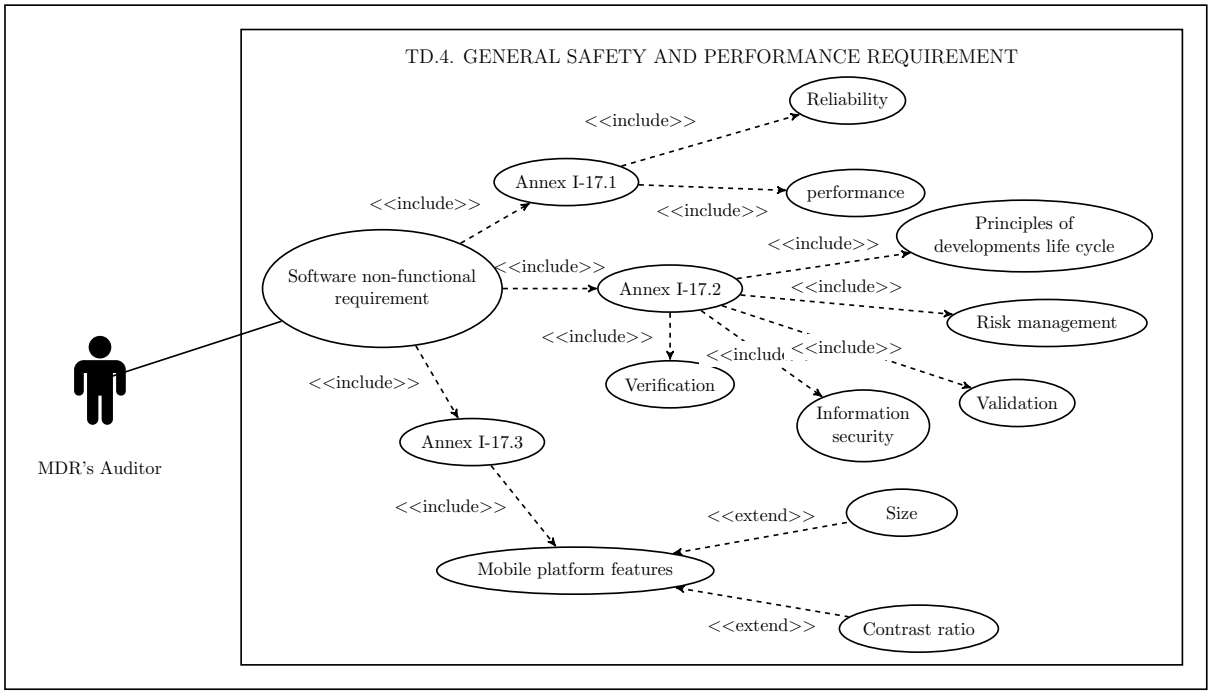

Figure 3.6: Use Case Diagram for Annex II 4.

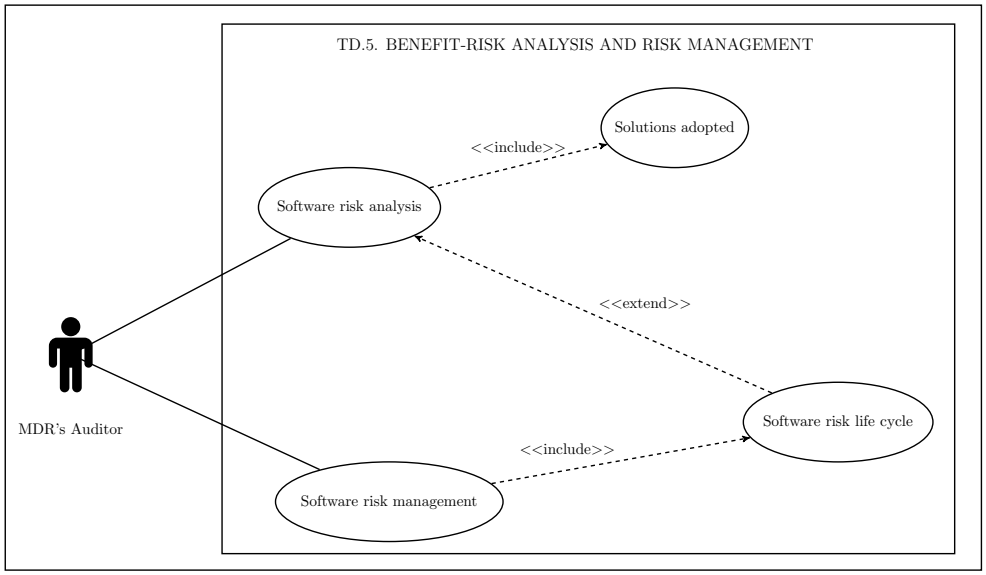

Figure 3.7: Use Case Diagram for Annex II 5.

3.8 Mapping between EU MDR requirements and Agile XP principles

The medical device software industry is a strictly regulated field, where development processes are long, development activities are precisely documented, user testing is only done towards the end of the development life cycle, and software cannot be released unless it fully complies with all requirements of medical devices regulations. The aim of this section is to identify the activities, principles, and procedures involved in Agile XP that can meet the extracted requirements from EU MDR. This mapping investigates the capability of Agile XP to meet EU MDR requirements. The mapping is conducted based on the challenges found in Chapter 2, as shown in Table 3.7.

Table 3.7: Mapping between EU MDR requirements and Agile XP practices.

EU MDR Requirements (Challenges)	**Supported by Agile XP Practices**	**Description**
Up-front Planning	Not supported	Requirements are changeable in Agile XP
Documentation	Partially supported	User story card considered to be the only document in Agile XP
Traceability	Not supported	Agile XP has no traceability analysis in any of the development phases
Formality	Not supported	Agile XP has practices that are performed informally, such as face-to-face communication

Up-front planning: EU MDR requires fixed up-front planning that contains system requirements and information about the development phases as stated in Article 23.1 Chapter III of Annex I, and in Annex II is the technical documentation. Moreover, EU MDR requires a risk management plan for each system, as stated in Section 3, Chapter I of Annex I. Agile XP have no fixed up-front planning as the Agile XP planning approach focuses on delivering the features of the product rather than project activities; furthermore, requirements can be changed at any time during the developing process. Therefore, Agile XP does not support these requirements of EU MDR.

Documentation:

EU MDR requires complete documentation that clarifies the software input, software requirement specifications, software risk management systems, developing processes and their validation, and software verification and validation. In addition, the document should clearly define the non-functional requirements such as repeatability, reliability, performance and security and the safety of the software. One of the main values of Agile XP is working software over comprehensive documentation. Agile XP focuses mainly on programming, and the main concept behind Agile XP is to have less documentation. Documentation in the Agile XP exists in some practices such as the user story card, which is considered to be the only document in Agile XP that discuses only the highest level structures of the software without mentioning the non-functional requirements. Therefore, Agile XP partially supports the documentation required by EU MDR.

Traceability:

Traceability analysis defines the relationship between the software development stages by tracing the requirements from planning to deployment. Traceability analysis is a vital activity during the SDLC and is required by the EU MDR regulations. For example, the EU MDR emphasises traceability analysis during the requirement analysis phase by requiring manufacturers to establish, implement, document and maintain a risk management system throughout the entire lifecycle of a device, requiring regular systematic updating as well as relationships and dependency between system requirements.

Since the requirements in Agile XP are changeable at any time during the development process, it is difficult to have a traceability analysis between requirements, design, code, and test cases. However, there are still some XP practices that can be used for traceability purposes, such as the ID of the user story, and the acceptance test card can also be used to trace the requirements. Therefore, Agile XP has no entire practices that account for having traceability analysis in any of the development phases.

Formality:

The EU MDR requires the manufacturers to have formal methods that focus on delivering products correctly, and any change to the requirements should be recorded in a formal specification.

In Agile XP, some practices are performed informally, for example, requirements are gathered from the customer in an informal way, such as face-to-face communication, or via the user story card. Therefore, Agile XP is not formal enough to support the formality required by the EU MDR.

3.9 Summary

Medical device software companies in Europe must comply with EU MDR requirements to obtain the CE marking. Software developers face difficulties in identifying and understanding the relevant requirements of EU MDR. This chapter has explained the EU MDR and its devices classification. In addition, we have extracted the requirements from EU MDR that have direct or indirect impacts on the SDLC. Moreover, we have detected the ambiguity in the extracted requirements of EU MDR and then presented a solution that could minimise the ambiguities based on the software engineering techniques.

The following comments can summarise the findings of this chapter:

- Medical devices regulations are rules or directives made and maintained by authorities.
- Medical device software organisations must obtain approval and comply with one of the healthcare regulations based on the marketing region.
- In the USA, the Food and Drug Administration (FDA) and Health Insurance Portability and Accountability Act (HIPAA) are applied. In the European Union, Medical Device Regulations (MDR) are applied to get CE marking.
- EU MDR requirements that have a direct or indirect impact on the development life cycle were extracted and analysed.
- Ambiguity in the extracted requirements of EU MDR was identified and minimised using software engineering techniques, such as user story cards and UML use case diagrams.
- Mapping was conducted between EU MDR requirements and Agile XP principles to investigate the capability of Agile XP to meet EU MDR requirements. The mapping was conducted based on the challenges that have been explained in Chapter 2 as shown in Table 3.7.
- The results of the mapping study show that Agile XP does not fully support the EU MDR requirements.

Chapter 4

Extending the Agile XP User Story to Meet EU MDRs

Despite the benefits that Agile XP user stories provide, the number of methods to assess and improve user story quality is limited. According to the Agile XP manifesto, the only documentation that XP has is the user story, a piece of card that contains the stakeholders' requirements, written in a natural language during an interview or conversation. An XP user story does not satisfy the requirements of EU MDR, therefore it cannot be submitted as compliance evidence to EU MDR.

This chapter will propose an extension to the Agile XP user story to enhance the conformity to EU MDR requirements for the planning phase. The extension will be based on five sub-processes derived from the ASM ground [75] model and SPICE such as ISO/IEC 25030:2019 and ISO/IEC 25010:2011. These sub-processes are: identifying user story suppliers, ensuring the formality of functional requirements, proposing semi-structured forms for non-functional requirements, prioritisation of user stories, and identification of user story dependency. These sub-processes are integrated with the Agile XP user story to enhance the ability to collect and represent the system requirements corresponding to EU MDR requirements that are mandatory for the CE marking certificate. To investigate the conformity of the extended XP user story to the EU MDR requirements, a mapping study was conducted to check the capability of the extended Agile XP user story in meeting the EU MDR requirements for the planning phase, as well as applying the extended Agile XP user story on case studies to check the applicability of the extended Agile XP user story.

4.1 Agile XP User story card

User stories are a textual notation widely used in agile software development to capture the system requirements [76]. An Agile XP user story is an informal notecard that describes system functionality written from the perspective of the end user [29]. In other words, it is a piece of paper that contains the customer requirements written in natural language terms to represent the customer/user view of the intended software. User stories usually contain three components, which are a short piece of text describing the user story, the conversation between stakeholders about the story, and acceptance criteria [77]. This user story is written, developed, and validated during the XP planning game by the customers, team members and stakeholders [78].

The Agile XP user story captures only the essential elements of requirements that describe who requires it, what the output of the requirement is, and sometimes justification is included of why this requirement is important [79]. The most commonly used format for the user story is: ***As a*** $<$ ***type of user*** $>$, ***I want*** $<$ ***some goal*** $>$ ***so that*** $<$ ***some reason*** $>$ [76, 77]. For example: As an HR Manager, **I want** to view a candidate's status **so that** I can manage their application process throughout the recruiting phases.

Software developers found that using user stories for a project has several advantages, such as focusing on the user rather than on tasks, enabling collaboration between the teams, emphasising verbal communication, being comprehensible by everyone, the right size for planning, and encouraging deferring details [29, 77].

4.2 EU MDR requirements for the Agile XP user story

Since the Agile XP user story is conducted during the planning phase, an analysis was conducted on the requirements that are demanded by the EU MDR that have direct or indirect impacts on the planning phase of the development process. The Technical Documentation set out in Annexes II, III of EU MDR [53] was selected as it has the requirements that have an impact on the planning phase of the SDLC. According to EU MDR technical documentation, EU MDR requires a formal and fixed plan for the development to be activated from planning to testing, as well as documented evidence on how risk analysis has been done and how requirements have been implemented, designed, and tested. Moreover, EU MDR demands documentation for traceability purposes. EU MDR requirements that are related to the planning phase can be classified into six categories: User type, Functional requirements, Non-functional requirements, Risk Management, Traceability, Formality, and Test Report. Table 4.1 shows the EU MDR requirements for each category.

Table 4.1: Categories of the EU MDR planning requirements.

Categories	Description (EU MDR requirements)
User type	Annex II 1.1: (a) product or trade name and a general description of the device including its intended purpose and intended users.
Functional requirements	Annex II 1.1: (d) Principles of operation of the device and its mode of action. (j) A general description of the key functional elements, e.g. its parts/components (including software if appropriate), its formulation, its composition, its functionality. k) a description of the raw materials incorporated into key functional elements.
Non-functional requirements	Annex II 1.1: (l) technical specifications, such as features, dimensions and performance attributes (g) An explanation of any novel features. Annex I – 17.1. ensure repeatability, reliability and performance in line with their intended use. Annex II 4. general safety and performance requirements
Risk management	Annex II 1.1: (f) the risk class of the device and the justification for the classification rule(s) applied in accordance with Annex VIII; Annex I- 17.2. Principles of the development life cycle, risk management, including information security, verification and validation. Annex II 5. benefit-risk analysis and risk management : The documentation shall contain information on (a) the benefit-risk analysis referred to in Sections 1 and 8 of Annex I, and (b) the solutions adopted and the results of the risk management referred to in Section 3 of Annex I.
Traceability	Annex II 3. (a) information to allow the design stages applied to the device to be understood; (b) Complete information and specifications, including the manufacturing processes and their validation, their adjuvants, the continuous monitoring and the final product testing. Data shall be fully included in the technical documentation;
Formality	Annex II 4. (d) The precise identity of the controlled documents offering evidence of conformity with each harmonised standard, CS or other method applied to demonstrate conformity with the general safety and performance requirements. The information referred to under this point shall incorporate a cross-reference to the location of such evidence within the full technical documentation and, if applicable, the summary technical documentation.
Test Report	Annex II 6. product verification and validation: The documentation shall contain the results and critical analyses of all verifications and validation tests. Annex II 6.1. (b) detailed information regarding test design, complete test or study protocols, methods of data analysis, in addition to data summaries and test conclusions regarding in particular: -Software verification and validation.

4.3 Mapping between Agile XP user story and EU MDR requirements

The mapping study was conducted based on the categories of the EU MDR planning requirements mentioned previously to check the capability of using Agile XP user stories when developing medical device software that must comply with EU MDR. Table 4.2 shows a summary of the mapping between Agile XP user story and EU MDR requirements.

Table 4.2: summary of the mapping between Agile XP user story and EU MDR requirements.

EU MDR requirements	**Agile XP user story**
User type	Exists in As type of user...
Functional requirements	Exists in I want some goals...
Non-functional requirements	Not supported
Risk Management	Not supported
Formality	Not supported
Traceability	Not supported

User type: EU MDR requires the identification of the intended purpose and intended users of the software as mentioned in Annex II Article 1.1.a. This requirement is fulfilled by the Agile XP user story, where the type of user is identified at the beginning. Functional requirements and non-functional requirements: An XP user story contains only a general explanation of software functions written from the perspective of the end user. The XP user story does not contain any further details of the functional requirements or non-functional requirements. The EU MDR, on the other hand, requires documented details of the key functional elements and specifications of the system being developed as well as an explanation of any non-functional features, as mentioned in Annex I in Article 17.3 and Annex II in Article 1.1(d), (j), (k), (l), and (g) of EU MDR.

Risk Management: as mentioned earlier, the XP user story is a piece of paper that contains the requirements written by the customer. There is no evidence showing that an XP user story can handle the risk management life cycle which contains three steps: identification, analysis, and evaluation of the risk. The EU MDR requires in the up-front planning the risk class of the device and risk management, including information security, verification and validation as mentioned in Annex I Article 17.2, Annex II Article 1.1(f), and Annex II Article 5 and Article 6 of EU MDR.

Formality: XP user stories are usually written in natural language and in an informal way such as during an interview or conversation with the customer. They are written in the format of three sentences of text: As a ¡ type of user ¿, I want ¡ some goal ¿ so that ¡ some reason ¿. Requirements in XP user story are expressed in the customer's terminology. In contrast, EU

MDR requires to have a formal and precise document that contains system requirements to be submitted as evidence of conformity to EU MDR.

Traceability: studies such as [80, 81] show that the agile team facing challenges in tracing the user stories cards as they are acting only as an initial step to start the development but not for the project progress. This due to the lack of formal record that shows the interdependencies between user stories and also to the changing of the requirements during the development process which make it difficult to trace the user stories. These user stories usually attached to story wall which runs the risk of being easily lost or damaged [7]. In contrast, EU MDR requires a formal record that shows the interdependency between requirements which would help the auditor to trace back the requirement to its origin as mentioned in annex II Article 3(a) and (b) of EU MDR.

4.4 Extension methodology

This section will propose an extension to the Agile XP user story to enhance the conformity to EU MDR requirements for the planning phase. The extension will be based on five sub-processes derived from Abstract State Machine (ASM) ground model and Software Process Improvement and Capability Determination (SPICE) such as ISO/IEC 25030:2019 and ISO/IEC 25010:2011. Figure 4.1 illustrates the combination of the ASM ground model (Section 4.5.1) and SPICE (Section 4.5.2) resulting in the extended methodology of an XP user study, (Section 4.6).

4.4.1 ASM ground model

Abstract State Machine (ASM) is a method that supports the integration of problem-domain-oriented modelling and analysis into the development cycle [75]. ASM offers two methods that support the SDLC. These methods are the ground model method for requirements capture, and the refinement method for turning ground models by incremental steps into executable code[75, 82].

The ASM ground model was defined by Börger [75] as *"blueprints" whose role is to "ground designs in the reality". They represent succinct process-oriented models of the to-be-implemented piece of "real world"*. The ASM ground model could assist in requirements capture, requirements inspection, and requirements traceability by solving three common problems in the SDLC [75, 83]. Firstly, language and communication problems, where there must be clear communication between the application domain and the world of mathematical models. Therefore, the descriptions of data-oriented applications, function-oriented applications, and control-oriented applications must be naturally expressed in a common language that can be understood by the involved parties (domain experts and software developers). For example, using model language

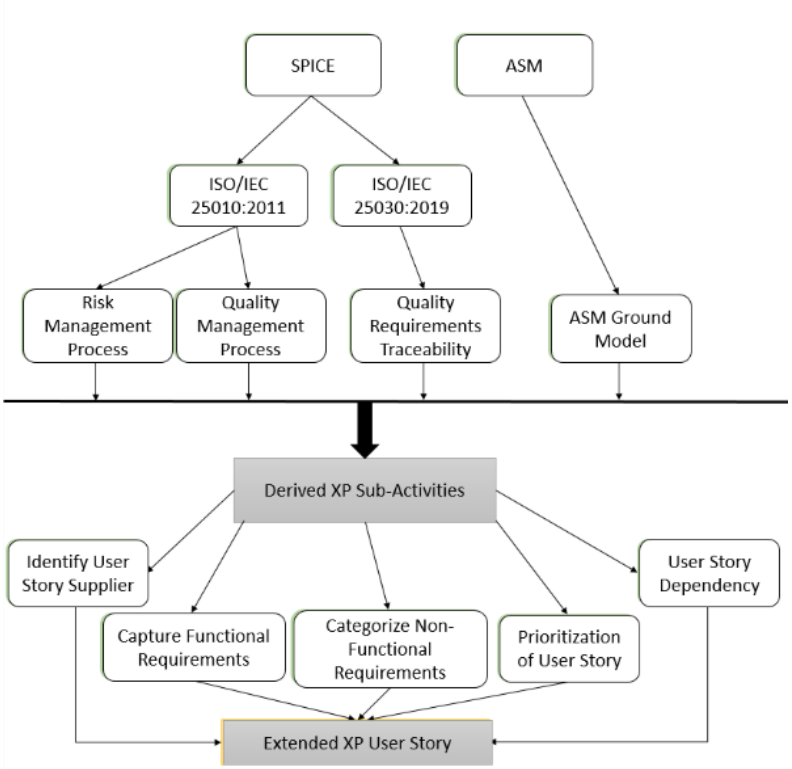

Figure 4.1: The extension methodology of XP user story.

to provide a general data model together with a function model and a suitable interface to the system would construct clear communication.

Secondly, the ASM ground model solves the problem of formalisation, which enables applications of formal verification methods to be applied on requirements capturing such as using mathematical means to present precise and consistent system requirements. The correctness of informal requirements cannot be proven without formalisation using mathematical means. While changes can be made in requirements for certain reasons, domain experts must inspect the ground model to check the completeness and consistency of the model. These two forms are the main aspects of ground model verification. To solve this problem, software developers could use the broad-spectrum algorithmic language of ASMs that enables the software developer to customise the ground model to resemble the structure of the real-world problem, enable its correctness to be checkable by an investigation, and its completeness analysable with regard to the problem to be solved.

Finally, the ASM ground model can help to solve the problem of validation. Ground model verification leads to validation problems. To have validation for software models, these models should be executable either conceptually or by machines. To solve this problem the ASM ground

model can be used for validation in two roles: as an accurate requirements specification and as a test model. The ASM ground model describes the operational steps of the system at the required level of abstraction, which comes as sets of basic natural language constructs (called rules) in the following form [82]:

If *condition* **then** *action*

Where a *condition* is any unambiguous expression describing a state or system and action is any unambiguous description of whatever kind of operations change the state of the system. The action has to be performed whenever the condition is true in the given state.

Properties of the ASM ground model:

To have a good ASM ground model, the following properties are required [82, 83]:

- **Precision:** where the requirements should contain no ambiguity that leads to confusion by the software developer.
- **Flexibility:** where requirements should be adaptable to different application domains.
- **Simplicity and concise:** where the requirements should be understandable by the domain experts and system users.
- **Resemblance:** where the requirements should resemble the structure of the real-world problem.
- **Abstraction:** where requirements should have only those aspects of the system's structure that affect the behaviour of the system being modelled.

4.4.2 SPICE

Software Process Improvement and Capability Determination (SPICE) is a project that combines international standards for software process assessment [84]. The aim of these standards is to improve the quality and productivity of the software development process as well as determining the process capability. SPICE standards that have been selected to be used in the extension methodology are:

- ISO/IEC 25030:2019: This provides the framework for quality requirements for systems as well as risk management processes [85].
- ISO/IEC 25010:2011: This provides quality characteristics and sub-characteristics of the requirements [86].

User story supplier: **User story ID:**

Priority:

Interdependency:

As < User story supplier > I want < if *condition* than *action* > **with Assumptions** < *Non-functional requirements* > **related to** < $Q1/\mathrm{Sub}1, \mathrm{Sub}\,2, \ldots, \mathrm{Sub}\ n\}$ *and Entity*{*Name of Entity* } >,< $Qn\{Sub1, Sub2, \ldots, Subn\}$, *and Entity* { *Name of Entity* } > **so that** < *some reason* >.

Figure 4.2: The extended XP user story format.

4.5 The Extended Agile XP User Story

As mentioned above, the sub-activities of the extended Agile XP user story have been driven by the ASM ground model and SPICE such as ISO/IEC 25030:2019 and ISO/IEC 25010:2011. Figure 4.2 shows the proposed extended format of the Agile XP user story. This section will explain each sub-activity of the extended Agile XP user story:

4.5.1 Identification user story supplier

EU MDR demands the identification of the intended users of the software. Identifying and understanding the sources of a user story would allow the software development team to identify, represent and manage the viewpoint of many different types of stakeholders[87]. ISO/IEC 25030:2019 defined a stakeholder as an individual or organisation that has an interest or share in a business or enterprise that meets their needs [85, 88]. Stakeholders include developers, testers, project managers, acquirers, independent evaluators, data owners, supporters, trainers, regulatory bodies and other people influenced by the system. Identifying the user story source could help in identifying the relationship between user stories, understanding the project problems, and identifying the system's actors and use cases. The identification of ¡User Story Supplier¿ of the extended Agile XP user story has been proposed based on the user classification of ISO/IEC 25030:2019 and EUMDR requirements as follows:

Customer supplier: requirements provided by the software customer, such as medical service organisations, healthcare experts, and patients. ISO/IEC 25030:2019 has classified the customer into three categories:

- Primary user: a customer who interacts with the system directly to achieve the primary goals of the system. For example, operators who upload, modify, and display data of patients by using the software.

- Secondary user: a user who interacts with the product to help the primary users to achieve the goal, such as content provider, system manager, administrator, security manager, maintainer, installer.
- Indirect user: a user who receives output from a system but does not interact with the system, such as the owner of the software and executive manager.

Development team supplier: anyone who was involved during the software development process can be considered as one of the development team, such as project managers, those responsible for managing the technical aspects of the project, developers, and those who design, implement, and test the software.

Authorities' supplier: anyone who demands requirements to ensure compliance with the regulations. This could be government requirements, healthcare regulations such as EU MDR requirements, EU MDR auditor, and EU MDR notified bodies.

4.5.2 Capturing functional requirements

Since the main ASM ground model principle is to have an abstract model by capturing only requirements that affect the behaviour of the software being developed. The requirement capturing process of the ASM ground model was integrated into the methodology of extending the Agile XP user story as shown in Figure 4.3. Capturing functional requirements can be done throughout ASM ground model requirements capture phases as follow:

Phase 1: abstractly describe the overall functionality of the system. This includes:

- Step1: write down the informal requirements from the user/customer.
- Step2: analyse the system elements (which have a direct correspondence in the model) and their properties by their signature, vocabulary, attributes, and relations.
- Step3: classify the system elements into functional requirements and their explicit assumptions (non-functional requirements).
- Step3: paraphrase the informal description to be modelled by using evocative names (terms that reveal intention).

Phase 2: identify the units in the requirements and parametrised variables for each unit.

Phase 3: add possible values to the variables.

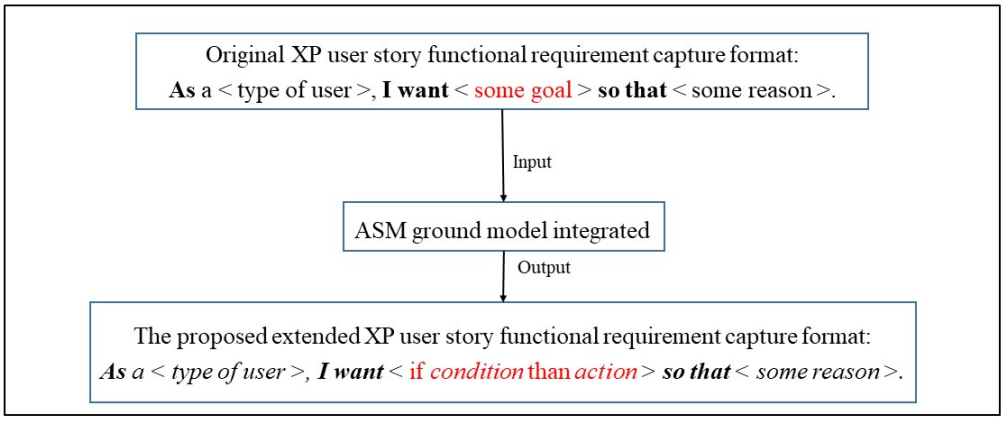

Figure 4.3: Integration of ASM ground model into the extended XP user story.

Phase 4: show the accurate actions of the system in an unambiguous way, by illustrating the conditions and actions of the system clearly.

The ASM ground model will assist the extended Agile XP user story in capturing the functional requirements by using the format of the ASM ground model which is < If condition then action >. In addition, this will help the developer to provide a high-level description of the requirements as well as precise and accurate requirements that can be submitted as evidence to the EU MDR auditors. Integrating the ASM ground model in capturing the functional requirements of the extended Agile XP user story would also help the developer to write the unit test at the same time. For example, the requirement for the user login page will be written as:

As a <primary user> **I want** < **If userId and Password = true Then grant access** > **so that** <I can login to the system>.

This means we have already provided the test criteria steps of the requirements, which are: first, enter User ID and Password and then click login, if login details are correct then login is successful.

4.5.3 Categorisation of non-functional requirements

Agile XP user story does not focus on non-functional requirements. When developing a critical software system, both functional requirements and non-functional requirements together should be considered [89]. Non-functional requirements (also called quality requirements) are software attributes that specify the operation of a system rather than functionality, such as performance, security, and reliability [71]. Non-functional requirements are a crucial aspect in developing critical software, such as medical device software, that should be addressed at the early stages of

Table 4.3: ISO/IEC 25010:2011Quailty Characteristics and Sub- Characteristics [86].

Characteristics	**Sub- Characteristics**
Functional suitability	Functional completeness, Functional correctness, and Functional appropriateness
Performance	Time behaviour, Resource utilization, and Capacity
Compatibility	Co-existence and Interoperability
Usability	Appropriateness recognisability, Learnability, Operability, User error protection, User interface aesthetics, and Accessibility
Reliability	Maturity, Availability, Fault tolerance, and Recoverability
Security	Confidentiality, Integrity, Non-repudiation, Accountability, and Authenticity
Maintainability	Modularity, Reusability, Analysability, Modifiability, and Testability
Portability	Adaptability, Install ability, and Replace ability

the software development process [90]. Therefore, EU MDR requires evidence showing that non-functional requirements have been considered during the development process and categorised based on their respective groups as mentioned in Annex I general safety and performance requirements, Article 17. ISO/IEC 25010:2011 has classified non functional requirements into eight characteristics, each with a set of related sub-characteristics as shown in Table 4.3. During the Agile XP planning phase, developers should identify the non-functional requirements, whether by gathering directly from the customer or by suggesting them to the customer, as the developers should have more knowledge about non-functional aspects of the software. These non-functional requirements will be categorised into one of the ISO/IEC 25010:2011 eight characteristics and their sub-characteristics.

In this sub-activity of the proposed Agile XP user story, we are integrating a semi-structured form into the XP user story based on the ISO/IEC 25010:2011 eight characteristics and the sub-characteristics to identify the non-functional requirements. At the beginning, the developer will specify to which quality characteristics and sub-characteristics the non-functional requirement belongs and then the developer should identify the target entity that would be affected by this non-functional requirement.

The proposed semi-structured form of categorizing non-functional requirements:

Assumptions <Non-functional requirements> **related to** <Q1 Sub1, Sub2,. . . , Sub n **and Entity**Name of Entity>, <Q nSub1,Sub2,. . . ,Subn, **and Entity**Name of Entity>.

Where:

- <Q1, Q2,.., Qn> : represent the quality characteristics.

- {Sub1, Sub2,. . . ,Subn}: represent the sub- quality characteristics.
- Entity {Name of Entity}: represent the target entity.

Example:

Assumptions <The customer must be able to access their account 24 hours a day, 7 days a week> **related to** < Q1: {Reliability}, Sub1: {Availability}, **and Entity**: {Database components} >.

The purpose of integrating this semi-structured form of non-functional requirements into an Agile XP user story is to keep the user story as lightweight as possible, and also to provide evidence to EU MDR auditors that the non-functional requirements have been considered in the planning phase of the Agile XP.

4.5.4 Prioritisation of User story

Requirements prioritisation is an important aspect in requirements decision and release planning [91], which is to give importance to the requirements to be implemented first. Requirements prioritisation is typically done according to two different approaches; prioritisation by implementation order and prioritisation by importance, where the priority of requirements is determined by the importance to some stakeholder such as personal preference,

avoidance, or risk [92]. However, for critical systems such as medical device software, the risk rate of the requirements is usually considered to be the main factor in determining the priority of the requirements [93]. EU MDR has classified medical device software into four categories based on the risk rate from low to high (see Chapter 3, Section 3.3.2). For this research purpose, the prioritisation method will be a risk-based approach, where priority requirements of each user story are determined based on the risk level of EU MDR medical devices software classification, where the higher risk is the highest priority. The prioritisation process of the extended Agile XP user story is a combination of two prioritisation techniques, as shown in Figure 4.4:

1. **Numerical assignment:** grouping requirements into different priority categories [94]. Therefore, requirements will be classified into three priority groups: critical, serious, and non-serious.
2. **MoSoCoW:** grouping requirements into four priority groups (Must, Should, Could, Would) [94].

Table 4.4 shows guidance for the extended Agile XP user story prioritisation to help the developer to identify exactly to which priority group the requirement belongs. One of the MoSoCoW words will be written in the user story form to represent the priority of the requirement. For example,

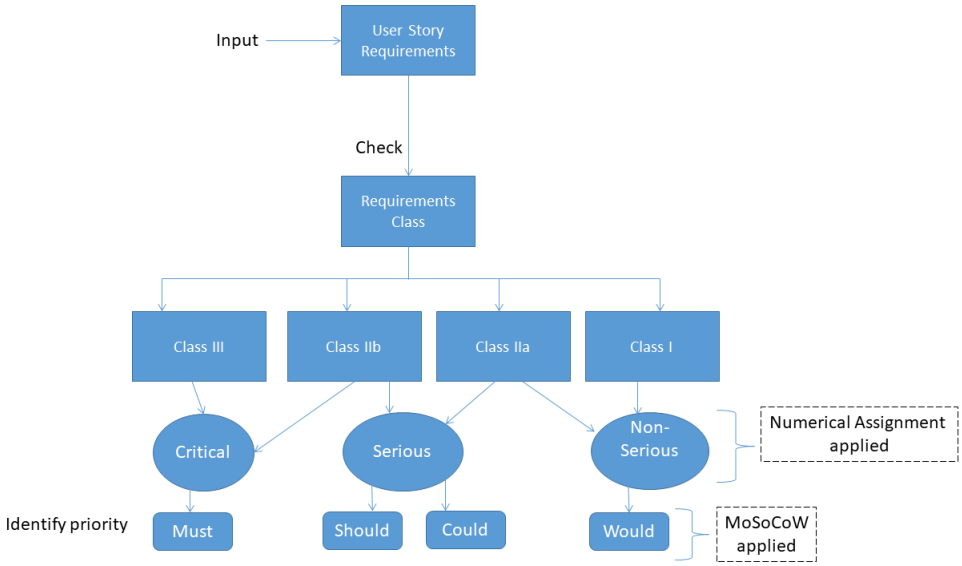

Figure 4.4: Requirements prioritization process of the expanded agile XP user story.

if the requirements in the user story are related to Class III the word "Must" will be tagged to the user story, meaning it has high risk and a very high priority to be developed first.

Table 4.4: Guidance for the extended Agile XP user story prioritisation.

	High risk	**Medium risk**	**Low risk**
Critical (Must)	Class III 1,2	Class IIb 1,2	Class IIa 1
Serious (Should)	Class IIb 3,4	Class IIa 2	Class IIa 3
Non-Serious (Could, Would)	Class IIa 4	Class I	Class I

4.5.5 User story dependency

User story dependency is a relationship that shows the traceability links between user stories during the development process. Actions performed on one user story could affect other user stories, therefore a user story cannot be treated independently [95]. Neglecting the interdependencies relations between user stories in the planning phase can have consequential effects on the development activities such as release planning, change management, design, implementation, and testing. For example, a change made for one requirement of a user story could lead to the change of other user stories as well. However, interdependencies relations between user stories should be identified in order to trace each requirement back to its origin. This process is known as requirements traceability, which is the ability to explain the life cycle of the requirement from its origin to the deployment in both forward and backward directions. When identifying the

Table 4.5: Interdependency types for an extended user story.

Type	Description
Refined_to	If US1 provides more details of US2 then US1 Refined_to US2. Or US1 breakdown into US1a, then US1 Refined_to US1a.
Change_to	US1 replaced US2 or new requirements added, then US1 Change_to US2.
Similar_to	US1 Similar_to US2 in terms of risk level, prioritisation and others. Then US1 Similar_to US2
Requires	US1 cannot be implemented before US2 is implemented, then US1 Requires US2
Before	Either US1 has to be implemented before US2 or vice versa, then US1 Before US2.
Conflicts_with	US1 cannot exist with US2 at the same time, then US1 Conflicts_with US2
Non-functional_of	When two user stories require the same non-functional requirements, US1 Non-functional_of US2.
Increase/Decrease _Risk_of	If one user story is selected to be implemented, then the risk of another user story increases or decreases. US1 Increase_Risk_of US2 or US1Decrease_risk_of US2.
Increase/Decrease _value_of	If one user story is selected to be implemented, then the value to the user story supplier of another user story increases or decrees. US1 Increases _value_of US2 or US1 Decreases_value_of US2

interdependency between user stories, some facts should be considered, such as prioritisation of the user story, non-functional requirements, database structure, and user story suppliers.

Interdependency types of the extended XP user story:

Interdependency describes the relationship that shows the types of the traceability links between user stories as shown in Table 4.5. Some of these interdependence types were described in [95, 96]. Most frequently used types were considered inhere with some modification to these interdependencies to fit our purpose, which is to make the extended Agile XP user story meet EU MDR requirements.

Refined_to: when developers break down a large user story into small user stories. These user stories each can be identified as Refined_to. For example, US1 breaks down into US1a, US1b, and US1c, then US1 Refined_to US1a. Or when more details are derived from user stories to other user stories.

Change_to: when requirements in user stories are changed by the developer or customer and a new user story replaced the old one. This dependency is identified as Change_to. For example,

US1 contains the original requirements and US2 contains the changed requirements, then US1 Change_to US2.

Similar_to: when one user story is similar to other user stories in terms of prioritisation and risk level. This dependency is identified as Similar_to. For example, US1 Similar_to US2.

Requires: when one user story (US1) cannot be performed until another user story (US2) is implemented, but not vice versa. For example, US1 cannot be implemented before US2 is performed, then US1 Requires US2.

Before: when one requirement implemented by the user story (US1) cannot be executed before the requirement is implemented by the user story (US2), because they are time-dependent. For example, either US1 has to be implemented before US2 or vice versa, then US1 Before US2.

Conflicts_with: when two or more user stories cannot be implemented at the same time or the implementation of one user story affects the implementation of another user story. The dependency here is identified as Conflicts_with. For example, US1 cannot exist with US2 at the same time, then US1 Conflicts_with US2.

Non-functional_of: when two user stories require or share the same non-functional requirements. For example, access security is a non-functional requirement for the login page for both user (US1) and admin (US2) requires that passwords shall never be viewable at the point of entry or at any other time. The dependency here is identified as US1 Non-functional_of US2.

Increase/Decrease_Risk_of: if one user story is selected to be implemented, then the risk of another user story increases or decreases. This dependency is identified as Increase/Decrease_Risk_of. This can be used as evidence of how the risk of each user story has been maintained. For example, if the user story contains a requirement stating the user must have a complicated password (a mix of letters, numbers and symbols randomly, and at least 15 characters long) for login. This will mostly decrease the risk of many other user stories, such as user stories of unauthorised access. And this can be written as US1 Increase_Risk_of US2 or US1Decrease_risk_of US2.

Increases/Decreases_value_of: if one user story is selected to be implemented, then the value to the user story supplier of another user story increases or decreases. This type of interdependency focuses on the effect relations between user stories may have on the user story suppliers' value. Some user stories have a positive or negative influence on the user story suppliers. To clarify, when a software developer suggests nice-to-have features to the customer which will lead to a positive influence for the customer, or by making functionality more complex, which will have a negative effect on the customer. And this can be written as US1 Increases _value_of US2

or US1 Decreases_value_of US2. For example:
US1: the web application should have an appointments calendar.
Nice-to-have US2: the calendar should synchronise this appointments calendar with the calendar of the mobile.
Then US2 Increases _value_of US1

4.6 Mapping between the Agile XP extended user story and EU MDR

The purpose of this mapping is to check the capability of the extended Agile XP user story in meeting the EU MDR requirements for the planning phase. Table 4.6 illustrates the mapping conducted between EU MDR requirements and the extended Agile XP user story. As a result, most of the EU MDR requirements for the planning phase exist in the proposed Agile XP user story. Therefore, the extended Agile XP user story could be considered as a fix to up-front planning when developing medical device software.

4.6.1 Applicability of the extended Agile XP user story

Extended Agile XP user story was applied to some case studies in order to prove the applicability of the extended Agile XP user story, such as the case study entitled 'SCD (Sickle cell disease) Web-based System for patients and clinicians' [97]. The purpose of this case study is to develop a unique web-based system to deliver remote monitoring for the SCD patients with a genetic blood disorder. This project was conducted in Liverpool John Moores University in collaboration with the department of haematology and oncology at Alder Hey Children's NHS foundation trust in the UK.

This case study presents the requirements in an informal way which cannot be submitted as evidence to the EU MDR. Therefore, requirements of the case study have been extracted and rewritten the requirements using the ordinary XP user story and then the extended XP user story was applied to make it more formal and precise. The authors of this case study detected the differences between the ordinary XP user story and the extended XP user story and they provided positive feedback regarding the extended Agile XP user story, as they have stated that using an extended XP user story in rewriting our SCD application requirements could help in achieving the CE mark certificate for our application.

Below is an example that illustrates the rewriting procedure of the requirements using the extended Agile XP user story for the case study entitled SCD (Sickle cell disease) Web-based System for patients and clinicians:

Table 4.6: Mapping between the agile XP extended user story and EU MDR.

MDR planning requirements	**Expanded XP user story**	**Description**
User Type	Identification of User supplier	Identifying and understanding the sources of software requirements User's supplier Development team's supplier Regulator's supplier
Functional requirements	Capturing Functional Requirements	User functional requirements written in form of ASM ground model: I want < if *condition* than *action* >
Non-functional requirements	Semi-structured of Non-functional requirements	Assumptions Non-functional requirements related to Q1 {Sub1, Sub2... Sub n} and Entity {Name of Entity}, Q n {Sub1, Sub2... Sub n}, and Entity {Name of Entity}.
Risk Management	Capturing functional requirements, Prioritization, and identification of user story dependency	• Since the priority assigned to the user story based on the risk level of the requirements, risk identification and analysis steps can be applied when assigning the priority to the user story. • User story dependency: Decrease/Increase_risk_of.
Traceability	Identification of User story dependency	Identification of user story dependencies: • Refined-to • Change-to • Similar-to • Requires • Before • Conflicts-with • Non-functional-of • Increase/Decrease-risk-of • Increase/Decrease-value-of
Formality	Using ASM ground model in capturing the functional requirements. Precis functional requirements and non-functional requirements.	ASM ground model for functional requirements: I want < if *condition* than *action* >. Assumptions <Non-functional requirements> related to <Q1 {Sub1, Sub2... Sub n} and Entity {Name of Entity}>, <Q n {Sub1, Sub2... Sub n}, and Entity {Name of Entity}>.
Test Report	Using ASM ground model in capturing the functional requirements	Using ASM ground model in capturing the functional requirements would help the developer to write the unit test at the same time. For example, the user login page will be written as If user ID and Password = true Then grand access. This means we have already provided the test steps of the requirements which are first, enter User ID and Password and then click login, if login details are correct then login successful

The original text of the requirement:

Patient side: Notification of high-risk condition of the patient.

The web-based application should send feedback messages to the medical specialists in association with high-risk conditions. The high-risk condition could happen when the number of heartbeats increases significantly. This information should be gathered regularly, anywhere and anytime, from patients who were diagnosed with SCD.

Ordinary XP user story:

As <web-based application> **I want** <Send feedback messages to the medical specialists in association with high-risk condition when number of heartbeats is significantly increasing > **so that** <doctors keep receiving information about the patient condition>.

Extended Agile XP user story:

User story supplier: customer **User story ID:** US1
Priority: Non-Serious (Could, Would).
Interdependency: US1 Requires US2
As < web-based application > **I want** < If heartbeats ≥160 bpm Then send alert to doctor> **with assumptions** < This information should be gathered from patients and alert should be sent anywhere, anytime, and on regular basis > **related to** < Reliability {availability} **and Entity** {Database} > **So that** < doctors keep receiving information about the patient's condition >.

4.7 Summary

The main contribution of this chapter is the extended Agile XP user story with the five sub-processes derived from the ASM ground model and SPICE standards such as ISO/IEC 25030:2019 and ISO/IEC 25010:201. The following comments show the advantages of the proposed extended Agile XP user story from the EU MDR perspective:

Formality: EU MDR auditors demand documented evidence at every phase of the development process to prove that processes are compliant with EU MDR requirements. For the planning phase, EU MDR requires formal and precise documented evidence that contains the intended users, system requirements, prioritisation, and risk management. The extended Agile XP user story will provide formal evidence that the intended user has been identified in <user story supplier>, each user story has been prioritised based on the risk rate, and the functional and non-functional requirements of the system have been documented formally using the ASM ground

model <if condition then action> with assumptions <non-functional requirements>. The extended Agile XP user story will present the system requirements in a precise, flexible and abstract way.

Non-functional requirements: the ordinary Agile XP user story does not focus on non-functional requirements. Where developing medical device software, both functional requirements and non-functional requirements together should be considered. The extended Agile XP user story will provide the non-functional requirements in Assumptions <Non-functional requirements> related to <Q1 {Sub1, Sub2,.. Sub n} and Entity {Name of Entity}>, <Q n{Sub1,Sub2,...,Subn}, and EntityName of Entity>.

Risk management: capturing the functional and non-functional requirements, prioritisation, and the identification of user story dependency can help in providing a risk management system at the early phase of the developing process. Traceability: using the interdependency types of the extended Agile XP user story will help in detecting the traceability links between user stories during the development process.

Test report: the extended Agile XP user story can provide a test report for each user story. Using the ASM ground model in capturing the functional requirements will help the developer to write the unit test and acceptance criteria in a formal way

Chapter 5

An Auditing Model for EU MDR Requirements in the Agile XP Environment

For medical device software organisations in Europe to get the CE marking, they must comply with EU MDR requirements. Compliance occurs once they submit evidence proving that their device development process has been accomplished in compliance with the EU MDR requirements. However, medical devices software companies that use Agile XP as their main development process face challenges in providing the auditing evidence. And this is due to the challenges that were analysed in Chapter 2, namely no fixed up-front planning, lack of documentation, traceability issues, and formality.

BS EN ISO 9000 defined auditing as systematic, independent examination, and documented the process for obtaining audit evidence and evaluating it objectively to determine the extent to which audit criteria are fulfilled [98]. In other words, auditing is the process of determining whether a software development activity meets the quality requirements of a specific regulatory body or standard, as well as to determine whether or not these activities are implemented effectively and are suitable to achieve the intended objectives. The focus of an audit can be a product, service, process or system of the organisation. There are two types of audits as classified by BS EN ISO 9000: an internal audit which is performed by the organisation itself and an external audit which is performed by parties having an interest in the organisation, such as customers or by a notified body, which is an external regulatory organisation that provides certification or registration of conformity with requirements such as those of FDA, ISO 9001, and EU MDR.

This chapter proposes the design methodology of an auditing model for EU MDR requirements to be used in Agile XP environments. The design methodology of the auditing model is

based on a framework of evaluation method proposed by Scriven [99]. The proposed auditing model consists of four common evaluation components: identification of the target, design audit criteria and yardsticks, evidence gathering and synthesis, and auditing decision. Moreover, the design of the auditing criteria and yardsticks could not be achieved without the support of quality management system standards such as BS ISO-IEC 12207, BS EN ISO 13485, BS ISO-IEC 25010, and BS EN ISO 14971.

The main goal of the proposed auditing model is to enhance the auditability of Agile XP in regard to EU MDR requirements. This goal can be achieved by assisting the EU MDR auditors to audit the development process of the medical device that has been developed using the Agile XP practices with regard to the EU MDR requirements in order to obtain evidence in conformity to EU MDR requirements. And also, this auditing can be considered as a guideline for the Agile XP developer to follow the EU MDR requirements.

5.1 The auditing process in the EU MDR

Most conformity assessment procedures of EU MDR consist of both the quality management system audit and the assessment of a device's safety and performance [100]. Usually, the auditing of EU MDR is performed by notified bodies, which are organisations accredited by EU MDR, they are responsible for auditing and certifying the manufacturer's quality management system as well as evaluating the conformity of certain products such as medical devices before being placed on the market.

In general, the auditing process consists of three activities: collecting evidence, comparing with criteria, and decision making. These activities are performed by auditors whether internal or external. Auditors collect evidence, which is defined as records, statements of fact or other information which is relevant to the audit criteria [98]. This evidence is then compared against the EU MDR auditing criteria to verify the proper implementation of the quality management system, and then the auditor decides whether or not the evidence complies with the regulations.

The EU MDR auditing process usually takes place at the site of the manufacture by an on-site auditor, who is personnel responsible for carrying out the audits of the manufacturer's quality management system. This auditing process is conducted in two stages:

Initial certification audit, stage 1 During this stage the documentation of the quality management system is reviewed and other items are checked to determine whether the manufacturer's product is ready to move on to stage 2.

Initial certification audit, stage 2 During this audit stage, the implementation and effectiveness of the quality management system of the product is evaluated. The findings of the audit are analysed by the audit team and the decision is written in an audit report that is submitted to

the organisation.

5.1.1 Analysis of EU MDR auditing requirements

The auditing requirements of the EU MDR that are required during the auditing process are represented by the quality management system as described in Chapter 1 of Annex IX [53]. The quality management system should be documented in a systematic way that shows quality programmes, quality plans and quality records. Article 10(9) of EU MDR describes the quality management system that should be documented and implemented by manufacturers, such as all device information covered by the quality management system, documentation of the manufacturer's quality management system, a documented description of the procedures, and documentation of the evaluation plan.

The auditing process of EU MDR is based on Annex VII of the technical documentation of EU MDR to determine whether or not the manufacturer meets the quality requirements of the EU MDR. Therefore, the technical documentation contains the requirements that should be fulfilled and submitted for auditing purposes. Below is a sample of the technical documentation requirements that affect the auditing process:

- The notified body shall test an adequate sample of the devices produced or an adequate sample from the manufacturing process to verify that the manufactured device conforms to the technical documentation.
- The notified body shall have documented procedures to test a product sample and test devices and technical documentation during audits, according to predefined sampling criteria and testing procedures, to ensure that the manufacturer continuously applies the approved quality management system.
- Ensure that the manufacturer complies with the documentation and information obligations laid down in the relevant Annexes.
- Gather sufficient information to determine whether the quality management system continues to comply with the requirements of this Regulation.

5.2 Design methodology of the auditing model

In general, any evaluation procedure requires criteria and evidence to confirm the existence of the criteria. The proposed auditing model can be considered as an evaluation procedure that requires criteria and evidence. According to Annex VII of EU MDR requirements, an organisation should design an auditing plan that clearly defines the objectives, criteria and scope of the audit. This

plan should adequately address and take account of the specific requirements for the devices, technologies and processes involved [53].

The design methodology of the proposed auditing model is based on the framework of the evaluation method proposed by Scriven [99]. This framework contains a common set of basic components of any type of evaluation procedure. Scriven [99] and Lopez [101] classified these components into six sections, as shown in Figure 5.1. These six components should exist when designing an evaluation procedure:

1. ***Target:*** The object under evaluation.
2. ***Criteria:*** The characteristics of the target which are to be evaluated.
3. ***Yardstick:*** The ideal target against which the real target is to be compared.
4. ***Data-gathering techniques:*** The techniques needed to assess each criterion under analysis.
5. ***Synthesis techniques:*** Techniques used to organise and synthesise the information obtained with the assessment techniques. The result of the synthesis is compared with the yardstick.
6. ***Evaluation process:*** A series of activities and tasks by means of which an evaluation is performed.

These common components of the evaluation procedure have been investigated, analysed and applied by several software engineering researchers such as [102], [101], and [103]. According to Lopez [101] these components are closely interrelated, and the evaluation process can be customised based on the target. Once the target is identified, the evaluation process is selected, and its criteria and yardstick must be identified for evaluation purposes. Data related to the target, such as documentation of the development process, should be collected using specific data-gathering techniques. This data is then assigned to one of the target criteria and compared against the yardstick by applying synthesis techniques. The results of the comparison will be considered in the evaluation decision.

The target under evaluation is the main consideration when selecting the particular type of method that best suits the project. Determining the components that are characteristic of the evaluation method depends on the evaluation method type. House [104] classified the evaluation methods based on the types of results that will be obtained. He classified evaluation methods into two categories: objective methods and subjective methods. The objective methods include objective-oriented evaluation, decision-making evaluation, needs-oriented evaluation, efficiency-oriented evaluation, and control-oriented evaluation.

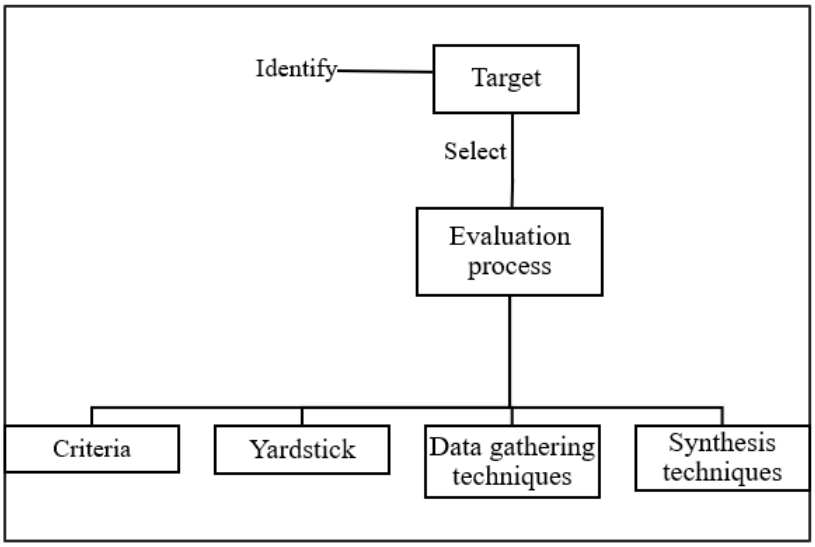

Figure 5.1: Common Components of an Evaluation Procedure.

5.3 Design of the proposed Auditing Model

Since the objective of the proposed auditing model is to ensure that the target is controlled and directed by the specified criteria and yardsticks, and the decision is made at the end of the auditing process to determine whether the target has met its goal, the evaluation method of the proposed auditing model can be considered as a type of hybrid approach that combines the principles of control-oriented evaluation and decision-making evaluation. Figure 5.2 illustrates the main process for designing an audit model for EU MDR requirements based on the evaluation framework described in [99] and [101].

5.3.1 Target identification

The first step in designing an evaluation procedure is to identify the evaluation target [103]. The target should be identified in order to identify the components of the auditing model such as criteria and yardsticks. In this research, designing an auditing model for EU MDR requirements in an Agile XP environment is the target, and achieving the EU MDR requirements is the main goal of the target. The process of target identification takes as input the factors derived by EU MDR requirements and Agile XP practices in identifying the target, which is the design of the auditing model.

The aim of this auditing model is to assist EU MDR software auditors to audit the Agile XP practices with regard to the MDR requirements to obtain evidence in conformity with EU MDR as well as helping the Agile XP developers to follow the EU MDR requirements.

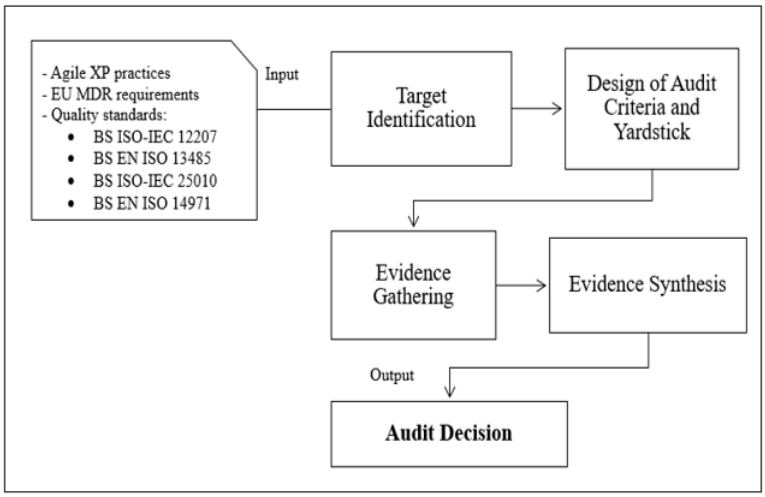

Figure 5.2: Design process of the proposed auditing model.

5.3.2 Design of the audit criteria and yardsticks

This is the essential step in developing an evaluation procedure. In this step the characteristics of interest for the target should be identified [103]. These characteristics are referred to as the auditing criteria. The election process of the auditing criteria can be based either on certain obligatory standards that contain implicitly the criteria to be applied in the evaluation or on diverse techniques for selecting the criteria, such as a functional analysis of the target and a needs assessment, for example a study of the needs, market preferences, values, standards, or ideals that might be relevant to the target [101].

In this auditing model, the obligatory standard is the EU MDR regulations. Since the EU MDR regulation does not contain explicit criteria for the process of the SDLC, the researcher has to find out other quality standards to support the election process of the criteria. The designing of the audit criteria cannot be achieved without support from other quality standards that contain requirements relevant to the quality management system and quality assurance management system, which have an impact on the SDLC. The selected standards are:

- BS ISO-IEC 12207: Systems and software engineering — Software life cycle processes [105].
- BS EN ISO 13485: Medical devices quality management systems — Requirements for regulatory purposes [106].
- BS ISO-IEC 25010: Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) —System and software quality models[86].
- BS EN ISO 14971: Medical devices — Application of risk management to medical devices [107].

Therefore, the auditing criteria were derived from EU MDR requirements and the quality standards mentioned above.

Lopez[101] proposed a diagrammatic tree that classified the criteria into two categories. First, general criteria, which are characteristics that cannot be assigned a value directly and need further analysis. Second, specific criteria, which are characteristics that can be assigned a value directly. In our proposed auditing model this criteria tree was used to classify the criteria and yardsticks of the proposed auditing model as shown in Figure 5.3. The description of the target

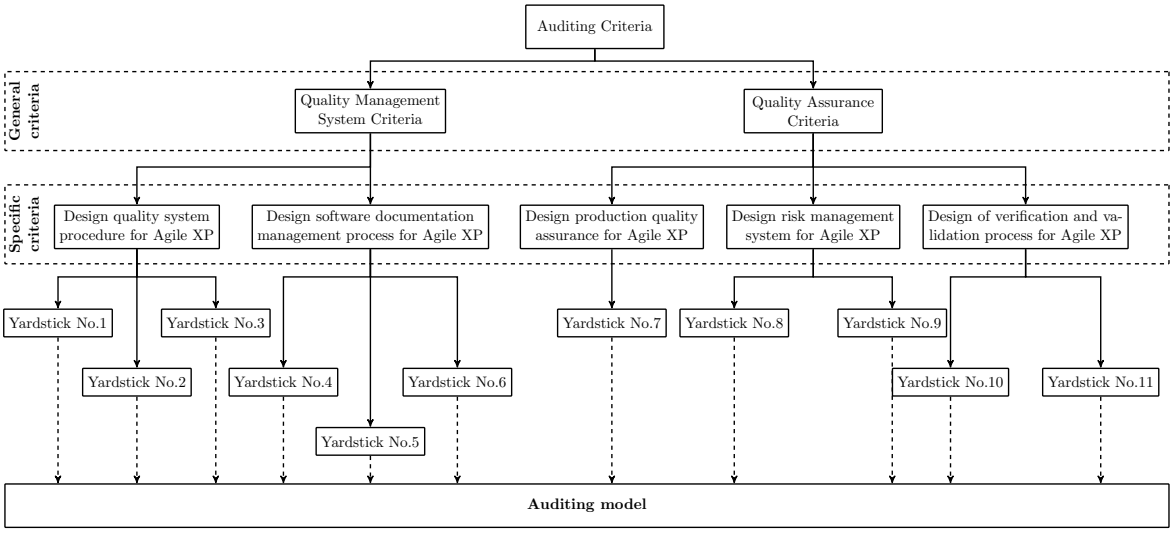

Figure 5.3: The structure of the proposed auditing model.

identification and criteria classification is the basis for developing the yardsticks for the auditing model. These yardsticks must contain the specifications, requirements, descriptions, or values for each criterion considered [101]. To reach the criteria, the yardstick must have threshold values to indicate the minimum value for each criterion [103]. Hence, for each criterion, there is more than one yardstick derived from the criteria based on the target considered in the auditing model.

5.4 The audit criteria and yardsticks

5.4.1 Quality management system criteria

The quality management system (QMS) is process-oriented and focuses on the development process such as process documentation and process review [108]. Moreover, QMS focuses on process-related issues such as audits and inspections, and regulatory compliance. The purpose of QMS is to ensure that the development process has been accomplished following specific regulatory requirements. The list of the audit criteria in this section is derived from EU MDR and supported by BS EN ISO 13485 and BS ISO-IEC 12207.

The main goal of designing the QMS criteria is to ensure the software development process is in compliance with the EU MDR requirements. QMS criteria include the following:

1. **Design Quality system procedure for Agile XP**
   An organisation should implement a QMS procedure to ensure compliance with the EU MDR regulation as well as to ensure that the QMS remains adequate and effective. Jim McCall's classification of quality requirements in Agile XP[109] and the BS EN ISO 13485 quality characteristics enable us to see what types of quality features have been used in the design of quality systems methods. Jim McCall classified software quality characteristics into three categories that contain 11 quality attributes [110]. Firstly, the software revision category, which consists of flexibility, maintainability and testability attributes. Secondly, the software operation category, which contains reliability, precision, effectiveness and usability attributes. Finally, the software conversion category, that consists of reusability, transferability and interoperability attributes.

   These features approximately meet the quality requirements of EU MDR that are specified in Annex II, technical documentation. Therefore, the quality system procedure of Agile XP should be based on the above quality attributes. Agile XP practices that we believe can fulfil the quality system procedures are planning game, user story, on-sit customer, and small release.

   The following are the audit yardsticks for these criteria; these auditing yardsticks indicate the minimum value for each criterion that must exist to submit evidence that the organisation has implemented a quality procedure during the development process:

   Yardstick No.1 ***(planning for quality system procedure)***

   Planning is an essential aspect in the quality system procedure, where software requirements should be collected and documented. XP user stories can be considered as the main planning part for the quality system procedure of the Agile XP development process, which can be seen in the exploration phase of the Agile XP, as it is the only document provided by the XP agile process. An Agile XP user story is a card that is written by the customer in collaboration with the developer. This card contains customer requirements, requirements specifications, acceptance tests, and estimation. By providing the extended XP user story that was proposed in Chapter 4, we will have a better-documented planning for quality system procedure.

   Yardstick No.2 ***(reviewing the requirements)***

   Small release practice assists in reviewing the requirements prior to supply or development. Each new iteration has the phase of iteration planning, which is to ensure that the require-

ments are reviewed and added incrementally as agreed between developers and customers. And this practice would improve the quality procedure of the Agile XP by satisfying the customer needs.

Yardstick No.3 ***(adapting to requirements changing)***

The involvement of the customer throughout the development process (on-site customer) would enable several questions to be answered directly from the customer, and this can improve the development planning of the system. Customer involvement would enable the developer to obtain earlier feedback from the customer regarding the ambiguity of the requirements, cost and time estimation, and other aspects.

2. **Design software documentation management process for Agile XP**
EU MDR demands documentation for each development phase, such as technical documentation that contains system requirements (functional and non-functional) and system specifications, as well as demanding documented evidence proving that EU MDR requirements have been applied throughout the system development stages. BS EN ISO 13485, 4.1.1standard clarified what kind of documentation is needed for the SDLC. Documentation includes documented procedures and records required by the regulation, records determined by the organisation to ensure the effective planning, operation and control of its processes, and other documentation specified by applicable regulatory requirements. BS ISO-IEC 12207 standard defined the purpose of the software documentation management process, which is to develop and maintain the recorded software information during the development process. These documents shall remain readable, unambiguous, and available. While Agile XP lacks the documentation, some practices can represent the documentation management process, such as the user story card, task card, collective code ownership, and tracker role.

   The following are the audit yardsticks for these criteria. These yardsticks are to ensure that the documentation management system does exist during the development process.

   Yardstick No.4 ***(documentation for software requirements and specifications)***

   Using some of the Agile XP practices and rules would enable the developer to have a documentation management system that could then be submitted to MDR auditors for auditing purposes; these practices are:

   - The user story card defines the functional requirements of the system to be implemented which are required by the customer, and also defines priorities and estimation for the next iteration.

- The task card is used by the developer during the iteration planning, where stories are converted into tasks. It is smaller than the whole story, and sometimes one task will support several stories. The task card consists of a story number, responsible developer, task estimation, task description, developer's notes, and task tracking [3].

These practices fully define the software functional requirements obtained from the customer. Moreover, an acceptance test shows how the system should function and what should be functional before accepting the software. The results of the acceptance test are recorded, whether success or failure, these records could then be submitted for auditing purposes.

Yardstick No.5 ***(documentation for source code)***

Collective code ownership is a practice of code sharing, where all the team can edit and view the code. In this practice, all the code is recorded, so that errors can be easily traced. That means code modification and reviewing is recorded and shared to all the development team. Besides that, the code of the unit test which is conducted by the developer also documents the behaviour of the software. These practices could assist in submitting the documentation for source code to the auditor.

Yardstick No.6 ***(documentation for traceability purposes)***

Refactoring and small release help to trace the changes of requirements. The tracker is one of the Agile XP roles, a person who is responsible for retrieving information about the evaluation to make sure that functionalities are implemented as estimated, and also responsible for keeping a log of all tests results, error and correction procedures. In addition, the use of a CRC card (class responsibility collaborator) which identifies and organises the object-oriented classes would help to trace the design phases. The CRC card is the only design work product produced as part of the Agile XP process [111].

5.4.2 Quality assurance criteria

Quality assurance can be defined as the part of quality management focused on providing confidence that quality requirements will be fulfilled [112]. EU MDR regulations demand evidence to ensure that product assurance quality characteristics have been fulfilled during the development process, as stated in Part 4 of Annex XI of EU MDR *'the manufacturer shall ensure that the QMS approved for the manufacture of the devices concerned is implemented and shall carry out a final verification'*. The list of the audit criteria in this section is derived from EU MDR (Annex XI, production quality assurance) and supported by BS EN ISO 13485 and BS ISO-IEC 25010

standards. The main goal of designing the quality assurance criteria is to ensure that that product assurance quality characteristics have been fulfilled during the development process. The quality assurance criterion is as follows:

1. **Design production quality assurance for Agile XP**
   Production quality assurance as defined by the BS ISO-IEC 25010 standard consists of eight characteristics: functional suitability, performance efficiency, compatibility, usability, reliability, security, maintainability and portability [86]. The audit yardstick for this criterion is as follows:

   Yardstick No.7 ***(identification of quality assurance characteristics)***

   Agile XP has built-in quality assurance characteristics such as on-site customer, refactoring, pair programming, continuous integration, and frequent tests (unit test and acceptance test). All these agile practices are to ensure that the developed software is either fault-free, or has a minimum number of faults.

2. **Design risk management system for Agile XP**
   Organisations should identify and record any risks and problems occurring during the development process. According to EU MDR in Annex I, manufacturers shall establish, implement, document and maintain a risk management system. The risk management system documentation shall contain risk class, risk analysis, the solution adopted and the results of the risk management. Risk management consists of four phases[113];

   - Identifying: where the list of the risks that threaten the system is recorded,
   - Analysing where risks are possible and the impacts are assessed,
   - Prioritising where priority is assigned to the risk based on the possibility and the impacts,
   - And the resulting monitoring, where significant risk is resolved and maintained.

   The Agile XP development process is intended to reduce the risk as much as possible. Agile XP has practices that can meet the requirements of the risk management system and can be considered as a risk plan. Such practices are acceptance test, unit test, pair programming, iteration, on-site customer, and refactoring. The following are the audit yardsticks for this criterion. These yardsticks ensure the existence of evidence that the risk management system has been applied during the development process.

   Yardstick No.8 ***(planning for risk management system)***

During the exploration phase of XP, the developer and customer outline the limitations and experiment to find out the risks involved. In addition, the agile principles of the test-driven development and acceptance tests would enable the detection and analysis of the risk in the early phase of the development. As stated in [114] "acceptance tests allow the customer to know when the system works and tell the programmers what needs to be done". The success or failure of these tests is recorded.

Yardstick No.9 ***(the process of risk management system)***

The Agile XP practices of continuous integration, where the customer can have the short release of the system to be tested, would enable risk issues to be discovered and adopted early. And pair programming (continuous code inspection) where two programmers review the codes continuously would enable the detected risks to be reviewed and maintained continuously. Moreover, refactoring of the code would help in removing any ambiguities and redundancy from the code.

3. **Design of verification and validation process for Agile XP**
An organisation should perform a verification process to ensure that the design specifications and development outcomes have fulfilled the customer requirements (output requirements met input requirements). EU MDR demands documentation that contains the results and critical analyses of all verifications and validation tests to demonstrate the conformity of the device with the intended requirements. As mentioned in EU MDR Annex ii Article 6 (b) *"detailed information regarding test design, complete test, methods of data analysis, in addition to data summaries and test conclusions"*. This information shall typically include the summary results of all verification, validation and testing performed during the development process. To have a clear vision of what is required for software verification, ISO 13485-2016 states that the organisation shall document verification plans that include methods, and acceptance criteria. The following are the audit yardsticks for this criterion. These yardsticks ensure the existence of evidence that the verification and validation have been considered during the development process.

Yardstick No 10. ***(Verification)***

Verification is the process of evaluating software at each development phase to ensure that it meets the intended requirements and the development process is on the right track of creating the final product [115]. Verification activities include design review, code inspection, and code walkthrough. These verification activities are covered by Agile XP practices:

Refactoring: where the developer restructures an existing body of code and modifies its internal structure without affecting the external behaviour [116]. This practice would

provide code inspections functionality.

Small release: During this practice, code and design specification is reviewed, where small releases are given to the customer to test and approve each increment of the system. This approval is recorded and marked as 'customer seen'. This approval by the customer verifies that each release is working correctly.

Pair programming: where two programmers sit together to work on the same code, one is writing the code and the other is reviewing it. This practice serves as a continual design and code review process.

Yardstick No 11. ***(Validation)***

Validation is the process of testing the software or its specifications at the end of the development to ensure that it meets its requirements [115]. Validation activities in Agile XP are all types of tests such as unit testing, integration testing, and user acceptance testing.

Unit testing: where the developer generates automated tests to test the functionality of each requirement. This practice would validate that each unit of the system is functioning correctly.

Continues integration: where the development team needs to keep the system code fully integrated at all times, not once or twice. This practice would help in catching any bugs to be fixed as well as to validate that the code is clear of bugs. Acceptance testing: helps in improving the validation process of Agile XP. Acceptance testing that occurs frequently is considered a dynamic quality assurance technique in Agile XP [87]. Acceptance tests are performed by the customer to prove that the functionality of requirements is implemented correctly as required.

5.5 Evidence gathering and synthesis techniques

Gathering evidence techniques should be identified as part of building the auditing model. The goal of identifying these techniques is to obtain information needed to judge the target during the synthesis. Three main evidence-gathering techniques are used in the software engineering field. These types are [101]:

- **Measurement:** involves the use of the appropriate measurement instruments or mechanisms.

- **Assignation:** for example, questionnaires, interviews (individual or groups), documentation inspection, and simple tests not involving metric applications.
- **Opinion:** techniques for getting subjective data on the criteria, such as observation.

The selection of evidence-gathering techniques depends on the auditing model components (target, criteria and yardsticks). An auditor would assign for each criterion the types of values specified in the yardstick and the possible evidence-gathering techniques, as shown in Table 5.1, an example of assigning evidence-gathering techniques to criteria. Once evidence is obtained

Table 5.1: Example of assigning evidence-gathering techniques to criteria.

Auditing Criteria	**Yardstick**	**Evidence-Gathering Techniques**
QMS Criteria		
Design quality system procedure for Agile XP	**Yardstick No.1** (*planning for quality system procedure*) **Yardstick No.2** (*reviewing the requirements*) **Yardstick No.3** (*adapting to requirements changing*)	Document review, observation, interview
Design software documentation management process for Agile XP	**Yardstick No.4** (*documentation for software requirements and specifications*) **Yardstick No.5** *(documentation for source code* **Yardstick No.6** (*documentation for traceability purposes*)	Document review, code review, test, UML diagrams
Quality Assurance Criteria		
Design production quality assurance for Agile XP	**Yardstick No.7** (*identification of quality assurance characteristics*)	Document review, observation, code review, log of test, UML
Design risk management system for Agile XP	**Yardstick No.8** (*planning of risk management system*) **Yardstick No.9** (*process of risk management system*)	Document review, analysis, code review, log of test result, test
Design of verification and validation process for Agile XP	**Yardstick No.10** (*verification*) **Yardstick No.11** (*validation*)	Document review, test, observation, code inspections

by applying the evidence-gathering techniques, the auditor moves to the next step, which is evidence synthesis. This involves conducting a comparison on the obtained evidence against the yardsticks to judge the target and get the result of auditing. This step can be accomplished by applying synthesis techniques. Lopez has classified two types of synthesis techniques that can

be applied in the proposed auditing model [101]:

Single value: a single datum (numerical or otherwise) is obtained as a result of the evaluation. This group includes combination methods. When these techniques are applied, a meaningful value scale is required for the datum obtained.

Multiple values: These techniques, for example, statistical techniques, criteria grouping, and datum-by-datum comparison with the yardstick, output more detailed information than single-value techniques. In our case, the multiple values technique is more suitable as the auditing process is based on target criteria and yardsticks.

5.6 ASM for the proposed Auditing model

Abstract State Machine (ASM) methodology has been used to represent the proposed auditing model in a more formal and executable manner, as well as to clarify precisely the process of the proposed auditing model. In addition, using the ASM methods would assist in converting the proposed auditing model to a semi-automated auditing model in the future. ASM is a method that supports the integration of problem-domain-oriented modelling and analysis into the development cycle [75]. ASM offers two methods that support the SDLC. These methods include the **ground model method** for requirements capture, and the **refinement method** for turning ground models into executable code by incremental steps [75, 82]. (For more details on ASM see Chapter 4).

The ASM model describes the operational steps of the system at a very high level of abstraction, which comes as sets of basic natural language constructs called rules in the form of **If** *condition* **then** *action* [82], where a condition is any unambiguous expression describing a state of system and action is any unambiguous description of whatever kind of operation changes the state of the system. ASM methods provide several benefits when applied to software system engineering [117]. Such benefits include producing unambiguous specifications about the features and behaviour of a system, early detection of errors in the design process, and added formal analyses methods such as validation and verification that assure the correctness of software requirements and guarantee the required software properties.

Figure 5.4 shows graphically the ASM ground model for the proposed auditing model. The auditing process starts by uploading the documents that are to be audited. Once documents are uploaded, the process of extracting evidence and classifying evidence type begins. After evidence classification completed, the auditing model moves to the next step, which is assigning the evidence to the relative yardsticks, and sometimes one piece of evidence is related to more than one yardstick. If the evidence fulfils the yardstick criteria, then make an auditing decision and notify that auditing is completed.

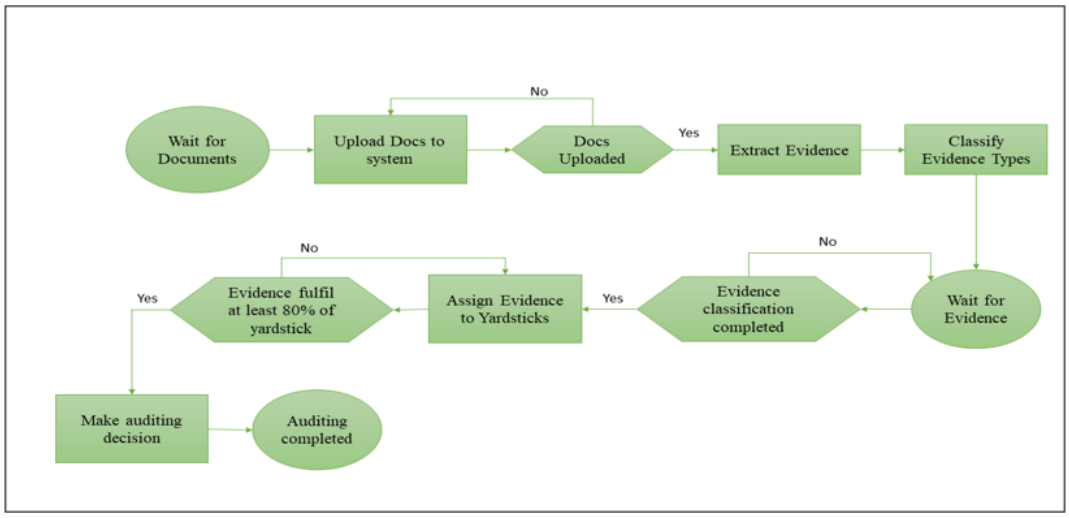

Figure 5.4: Graphical ASM ground model for the proposed auditing model.

5.6.1 AsmetaL

We have used one of the ASM implementations, the ASMETA, to present the proposed auditing model in a formal and executable manner. ASMETA is a toolset that is used to write ASM models in a textual and human-comprehensible form by using the concrete syntax AsmetaL [118]. AsmetaL is a textual notation to be used by modellers to effectively write ASM models within the ASMETA framework [119]. AsmetaL is a language that can be divided into four parts [119]:

- Structural language: which provides the constructs required for describing the structure of an ASM model.
- Definitional language: which provides the basic definitional elements such as functions, domains, rules and axioms characterising an algebraic specification.
- Language of terms: which provides all kinds of syntactic expressions which can be evaluated in a state of an ASM.
- Behavioural language or the language of rules: which provides a notation to specify the transition rule schemes of an ASM.

In AsmetaL an ASM model is structured into four sections: a header, a body, a main rule and an initialisation [119]:

- Header: contains names of the imported modules, domains, functions, and rules which are imported from the module or exported from the ASM. And also the signature, which contains declarations of domains and function used in the ASM.

- Body: consists of definitions that contain declarations of static concrete domains, derived functions, terms, variables, and transition rules.
- Main rule: the core rule of the program that is to be executed.
- Initialisations: contain the initial states of the ASM. For the ASM executable code for the proposed auditing model see Appendix B

5.7 Summary

The main contribution of this chapter is the proposed auditing model for EU MDR requirements to be used in Agile XP environments. The design methodology of the auditing model was based on the framework of evaluation methods proposed by Scriven. The proposed auditing model consists of four common evaluation components: identification of the target, design audit criteria and yardsticks, evidence gathering and synthesis, and auditing decision. In addition, the design of the auditing criteria and yardstick was supported by quality management standards such as BS ISO-IEC 12207, BS EN ISO 13485, BS ISO-IEC 25010, and BS EN ISO 14971.

The main goal of the proposed auditing model is to enhance the auditability of Agile XP in regard to EU MDR requirements. The proposed auditing model could assist the EU MDR software auditors to audit the medical device developed by the Agile XP practices with regard to the EU MDR requirements in way of obtaining evidence in conformity to EU MDR requirements. In addition, the proposed auditing model could act as a guideline that allows the Agile XP developer to follow the EU MDR requirements. Moreover, ASM was applied to represent the proposed auditing model in a more formal and executable manner as well as to clarify precisely the process of the proposed auditing model.

Chapter 6

Evaluation of the Proposed Auditing Model

An evaluation can be defined as the process of determining the performance, quality, value, and importance of the target[99]. The purpose of the evaluation in this chapter is to investigate the applicability of the proposed auditing model that has been described in Chapter 5. The evaluation was performed based on the planning, examination, decision-making (PED) evaluation framework[101]. Seven case studies were selected to be audited by applying the proposed auditing model to investigate whether or not they comply with the EU MDR requirements. This chapter will explain the evaluation procedure of the proposed auditing model and the auditing results of the selected case studies.

6.1 Evaluation procedure of the proposed auditing model

In order to investigate the applicability of the proposed auditing model, the evaluation framework of planning, examination, and decision-making (PED) was used. This evaluation framework has been investigated and used by [101, 103]. This framework contains the main phases of evaluation as follow:

Planning or preparation: This phase involves preparing the data to be audited by the proposed auditing model, including gathering case studies to be evaluated and contacting the medical device software organisations in order to apply the auditing model in the real field. This phase ended when all the evaluation components were gathered.

Examination: During this phase, we applied the evidence-gathering techniques that were been explained in Chapter 5. This phase ended when all the evidence was obtained for all the auditing criteria and yardsticks.

Decision making: During this phase, we applied the evidence synthesis techniques that were explained in Chapter 5. This phase ended when the comparison of the obtained evidence against the yardsticks was completed.

6.2 Selection of the case studies

To find the most relevant case studies, the author conducted a search through scientific journal search engines such as Springer, Emerald, Elsevier, IEEE, as well as the databases of medical journals, for example, *BioMed*, *Telemedicine*, and the *Journal of Biomedical Informatics*.

Keywords searched included "Agile XP and EU MDR regulations", "Agile XP in the healthcare industry", "UK healthcare software projects", "electronic health record using Agile XP", "Europe medical devices projects", "COVID-19 UK software implemented by Agile XP".

The selection process of case studies was based on several aspects such as:

- ***Quality of the case study:*** The case study should be reliable, and the findings should be of high quality, for example, a well-documented controlled experiment with industrial participants conducted by researchers.
- ***Relevance of the case study:*** The case study should be relevant to this research goal, for example, a case study that combines EU MDR and Agile XP practice in the medical device industry, or a case study providing details about medical device software implemented using Agile XP.
- ***Age of the case study:*** The case study should be up to date with technology changes.
- ***Vested interest of the case study:*** The medical device industry should have a vested interest in the outcome of the case study, for example, a controlled experiment, a single case study, a cross-company survey, well documented and published in a peer-reviewed conference or journal.
- ***Strength of the case study:*** based on the above aspects, the author identify the strength of the selected case studies if they are high-quality, relevance, updated with technology, and have vested interest.

6.3 Auditing evidence

In auditing financial systems, audit evidence refers to information (such as counting records, internal and external documents) used by the auditor in arriving at the conclusions on which the auditor's opinion is based [120]. In software engineering, evidence can be considered as data

supporting the existence or verity of something [86]. These data could be documentation for traceability purposes, evidence of verification, preventive action, and corrective action. This evidence could be obtained by observation, measurement, or test.

In a regulatory environment, audit evidence can be considered as records, statements of fact or other verifiable information relevant to the audit criteria [98]. EU MDR defined evidence as data and evaluation results pertaining to a device of a sufficient amount and quality to allow a qualified assessment of whether the device is safe and achieves the intended benefits when used as intended by the manufacturer [53].

Regulatory auditors usually look for evidence to demonstrate conformity to the regulations. For example, EU MDR auditors look for evidence of compliance with the general safety and performance requirements set out in Annex I. This evidence could be obtained through the results of all verification, validation and testing performed by users or systems designed for their intended use. The EU MDR auditor shall verify that the obtained evidence and the auditing criteria are adequate and shall verify the conclusions drawn by the manufacturer on their conformity with the relevant general safety and performance requirements.

6.3.1 Types of auditing evidence

Qasaimeh [102] has classified the auditing evidence of information systems into three main categories, which are textual evidence, modelling evidence, and graphical evidence. In this research an additional category was added, which is the coding evidence. The four main categories of evidence types are described below:

Textual evidence: This means evidence presented in text format, such as system requirements, system specifications, user stories, a written conversation between customer and developer, any system explanations by the developer, and test documentation. This would provide a readable text to the auditors.

Modelling evidence: This includes evidence that is presented in a specific modelling format, such as Unified Modelling Language (UML), Abstract State Machine (ASM), Business Process Modelling Notation (BPMN), and Service-Oriented Modelling Framework (SOMF). This would provide the auditor with formal information on the system design.

Graphical evidence: Evidence presented in specific graphical format, such as line graphs, bar graphs, pie charts, photographs (e.g. screenshots) and burn down charts can provide the auditor with visual information about the system's low level clarification or identify the outcome of the system design.

Coding evidence: Evidence that is presented in a coding format, such as system source code,

testing code, syntax highlighting (e.g. HTML syntax highlighting), and code comments provides the auditor with a readable code to understand the structure of the system.

6.4 Auditing of the selected case studies

6.4.1 Case 1

Title: SCD web-based System [97]

Author: Khalaf, M

Purpose: This project was implemented as part of the PhD research project entitled "Machine learning approaches and web-based systems in the application of disease-modifying therapy for sickle cell". This project was held in Liverpool by John Moores University in collaboration with Alder Hey Children's hospital. The main aim of this PhD research was to develop a complete hospital system for the Department of Haematology and Oncology at the Alder Hey Children's NHS Foundation Trust, Liverpool, UK. The purpose of developing an SCD web-based system was to deliver remote monitoring for SCD patients with this genetic blood disorder. When the system detects any critical condition from the patient, it generates an automatic message to the medical doctors to provide support for optimal decisions. This web-based system was developed using the principles of the Agile XP software development process.

Auditing findings A summary of the auditing findings is shown in Table 6.1, with more details described below.

Table 6.1: Summarised auditing findings for case 1.

Case Title: SCD Web-based System				
Auditing Criteria and Yardsticks	**Evidence Existence (Yes/No/Partially)**	**Evidence Type**	**Evidence Location**	**Page Number**
Design Quality system procedure for Agile XP				
Yardstick No.1 (*planning for quality system procedure*)	Yes	Textual evidence and Graphical evidence	7.2 System Architecture. 7.2.1 Front-End and Back-End System.	158 161

Yardstick No.2 (*reviewing the requirements*)	Partially	Textual evidence	7.2.5 SCD Clinicians Web-based System Appendix B: Ethical approval certificate (HRA letter)	170 210
Yardstick No.3 (*adapting to requirements changing*)	Partially	Textual evidence	Appendix B: Ethical approval certificate (HRA letter)	210
Design software documentation management process for Agile XP				
Yardstick No.4 (*documentation for software requirements and specifications*)	Yes	Graphical evidence and Textual evidence	Figure 7-1: The Web-based proposed System. Figure 7-2: Front-end and back-end architecture. 7.2.2 Central Database. Figure 7-5: Log-on main page. 7.2.4 SCD Patient Web-based System. Figure 7-7: Patient's dashboard. Figure 7-9: Patient's symptoms platform. Figure 7-10: Patient information.	160 162 162 165 166 167 169 170
Yardstick No.5 *(documentation for source code)*	Yes	Coding evidence	Appendix D: Some MATLAB Code and PHP with HTML	212
Yardstick No.6 (*documentation for tractability purposes*)	Yes	Modelling evidence	Figure 7-3: Database schema of Web-based tables	163
Design production quality assurance for Agile XP				
Yardstick No.7 (*identification of quality assurance characteristics*)	Yes	Textual evidence and Graphical evidence	7.2.1 Front-End and Back-End System Figure 7-12: Patient's information platform	161 172
Design risk management system for Agile XP				

Yardstick No.8 (*planning of risk management system*)	Yes	Textual evidence and Graphical evidence	7.2.3 Security and Privacy Figure 7-4: Login table for patients	163 164
Yardstick No.9 (*process of risk management system*)	Yes	Textual evidence and Graphical evidence	Figure 7-8: Line graph representation. 7.2.4 SCD Patient Web-based System.	168 167
Design of verification and validation process for Agile XP				
Yardstick No.10 (*verification*)	Yes	Textual evidence and Graphical evidence	Figure 7-6: SCD patient web-based system. Figure 7-7: Patient's dashboard. Figure 7-10: Patient information. Appendix A: Training and Testing for Ensemble Classifier	166 167 170 204
Yardstick No.11 (*validation*)	Yes	Textual evidence and Graphical evidence	Figure 7-5: Log-on main page (patient and clinician) Figure 7-9: Patient's symptoms platform Figure 7-13: Dynamic blood test samples results. 7.3.2 Decision Support Systems in Health Care. Appendix C: completion letter.	165 169 172 175 211

Yardstick No.1 (*planning for quality system procedure)*: This yardstick investigates the existence of the quality system procedure, looking for any evidence of planning for the quality system procedure. Case 1 supports textual and graphical evidence for this yardstick. More precisely, this evidence is found in Section 7.2 System Architecture and Section 7.2.1 Front-End and Back-End System. This evidence provides a description about the exploration phase of the system development.

Yardstick No.2 (*reviewing the requirements*): This yardstick investigates whether there was any

review of the system requirements during the development process. The review of the requirements can be done by the developer in collaboration with the customer at the beginning of each iteration. Case 1 partially supports this yardstick for textual evidence. Although there was some evidence of reviewing the requirements during the customer involvement in Section 7.2.5 SCD Clinicians Web-based System and in Appendix B: Ethical approval certificate (HRA letter), there was no precise evidence to show the feedback received from the customer. Therefore, more evidence is needed to show the participation of the customer during the development process.

Yardstick No.3 (*adapting to requirements changing*): This yardstick investigates whether there is any adaptation to the requirements change during the development process when the customer gives early feedback or new requirements. Case 1 partially support this yardstick for textual evidence in Appendix B: Ethical approval certificate (HRA letter). However, there was no precise evidence to show that customer has given new requirements to the developer or the developer has received early feedback regarding the requirements. Therefore more evidence is needed.

Yardstick No.4 (*documentation for software requirements and specifications*): This yardstick investigates the existence of documentation for system requirements and specifications. Case 1 fully support this yardstick for graphical evidence and textual evidence. The graphical evidence was in Figure 7-1: The Web-based proposed System, Figure 7-2: Front-end and back-end architecture, Figure 7-5: Log-on main page, Figure 7-7: Patient's dashboard, Figure 7-9: Patient's symptoms platform, and Figure 7-10: Patient information. The textual evidence located in 7.2.2 Central Database and 7.2.4 SCD Patient Web-based System. These texts contain descriptions of the system requirements to be implemented.

Yardstick No.5 (*documentation for source code*): This yardstick investigates the existence of source code documentation such as code modification, and reviews should be recorded to be submitted as evidence. Case 1 fully support this yardstick for coding evidence. This evidence is located in Appendix D: Some MATLAB Code and PHP with HTML.

Yardstick No.6 (*documentation for traceability purposes):* This yardstick investigates the existence of the tractability documentation. The focus here is on documents that show the tractability of the system requirements and design phases. Case 1 fully supports this yardstick with modelling evidence. This evidence is located in Figure 7-3: Database schema of Web-based tables.

Yardstick No.7 (*identification of quality assurance characteristics*): The focus of this yardstick is on the existence of quality assurance characteristics. This yardstick investigates whether the quality assurance characteristics were taken into consideration during the development process, such as functional suitability, performance efficiency, compatibility, usability, reliability, security, maintainability and portability. Case 1 fully supports this yardstick for textual evidence and graphical evidence. More precisely, this evidence is located in 7.2.1 Front-End and Back-End System and Figure 7-12: Patient's information platform.

Yardstick No.8 (*planning of risk management system*): This yardstick investigates whether there is any planning for the risk management system, to check whether system risks have been identified in the early developing phase. Case 1 fully support this yardstick for textual evidence and graphical evidence in

7.2.3 Security and Privacy and Figure 7-4: Login table for patients.

Yardstick No.9 (*process of risk management system*): this yardstick investigates the process of the risk management system to check that the risks have been detected, reviewed, and maintained continuously. Case 1 fully support this yardstick for textual evidence and graphical evidence in Figure 7-8: Line graph representation and 7.2.4 SCD Patient Web-based System.

Yardstick No 10. (*verification*): This yardstick ensures that the system meets the intended requirements and the development process is on the right track. Case 1 fully support this yardstick for textual evidence and graphical evidence in Figure 7-6: SCD patient web-based system, Figure 7-7: Patient's dashboard, Figure 7-10: Patient information, and Appendix A: Training and Testing for Ensemble Classifier.

Yardstick No 11. (*validation*): This yardstick investigates that the system meets the required specifications and requirements. The validation process is usually accomplished by testing the software. Case 1 fully supports this yardstick for textual evidence and graphical evidence in Figure 7-5: Log-on main page (patient and clinician), Figure 7-9: Patient's symptoms platform, Figure 7-13: Dynamic blood test samples results, 7.3.2 Decision Support Systems in Health Care, and Appendix C: Completion letter.

Auditing decision

After applying the proposed auditing model to this case study, it has been found that this case study fully supports nine auditing yardsticks out of 11. Two auditing yardsticks were partially supported and needed more evidence to be provided. For this case study, the author of this thesis worked in collaboration with the SCD Web-based System project authors to rewrite the requirements of the project in the proposed extended user story, as described in Chapter 4. Therefore, this case study would fulfil most of the yardsticks of design quality system procedure criteria, as the extended user story would provide evidence that there was planning for the quality procedure.

6.4.2 Case 2

Title: COVID-19 Care – A mobile application to help connect volunteers and vulnerable people in the community during the COVID-19 lockdown [121]

Authors: Khoa Phung, Silas Odongo and Emmanuel Ogunshile.

Purpose: They developed a mobile application that allows Covid-19 vulnerable people who are self-isolating to ask for help from the community in a safe manner. This project was designed and implemented to support the NHS of the UK in providing remote social care for people in self-isolation. Since they were required to deliver a minimum viable product within seven days and the size of the development team was small, they decided to use Agile XP as their primary SDLC. The development was fulfilled within two weeks.

Auditing findings

A summary of the auditing findings is shown in Table 6.2, with more details below

Table 6.2: Summarised auditing findings for case 2.

Case title: COVID-19 Care – A mobile application to help connect volunteers and vulnerable people in the community during the COVID-19 lockdown				
Auditing Criteria and Yardsticks	**Evidence Existence (Yes/No/Partially)**	**Evidence Type**	**Evidence Location**	**Page Number**
Design Quality system procedure for Agile XP				
Yardstick No.1 (*planning for quality system procedure*)	Yes	Textual evidence	V.REQUIREMENTS ELICITATION	3
Yardstick No.2 (*reviewing the requirements*)	Yes	Graphical evidence	TABLE III. C-19-C USER ACCEPTANCE TEST RESULTS	7
Yardstick No.3 (*adapting to requirements changing*)	Yes	Graphical evidence	TABLE III. C-19-C USER ACCEPTANCE TEST RESULTS	7
Design software documentation management process for Agile XP				
Yardstick No.4 (*documentation for software requirements and specifications*)	Yes	Graphical evidence, Textual evidence and Modelling evidence	TABLE I. C-19-C USE CASES. VI. B. Application implementation. Figure 1. High-level use case diagram.	4 5 4
Yardstick No.5 *(documentation for source code)*	No			
Yardstick No.6 (*documentation for tractability purposes*)	Partially	Textual evidence	B. User acceptance tests	6
Design production quality assurance for Agile XP				

Yardstick No.7 (*identification of quality assurance characteristics*)	Yes	Textual evidence and Graphical evidence	B. User acceptance tests TABLE II. C-19-C TEST CASES	6
Design risk management system for Agile XP				
Yardstick No.8 (*planning of risk management system*)	Yes	Textual evidence	V.REQUIREMENTS ELICITATION VII. D. Limitations with plausible solutions	3 8
Yardstick No.9 (*process of risk management system*)	Yes	Graphical evidence	TABLE II C-19-C TEST CASES. TABLE III C-19-C USER ACCEPTANCE TEST RESULTS.	6 7
Design of verification and validation processes for Agile XP				
Yardstick No.10 (*verification*)	Yes	Graphical evidence	Figure 2. Login screen. Figure 3. Signup screen. Figure 4. Phone verification Screen. Figure 5. NH user main screen. Figure 6. WH user main screen. Figure 7. Help request details - owner view. Figure 8. Help request details - public view	5 6
Yardstick No.11 (*validation*)	Yes	Graphical evidence	TABLE II. C-19-C TEST CASES TABLE III. C-19-C USER ACCEPTANCE TEST RESULTS.	6 7

Yardstick No.1 (*planning for quality system procedure*): This yardstick investigates the existence of the quality system procedure, looking for any evidence that there was any planning for quality system procedures, such as a user story that contains the system requirements. Case 2 supports this yardstick for

textual evidence. More precisely, this evidence is located in Section V. REQUIREMENTS ELICITATION. This evidence provides a description of the exploration phase of the system development.

Yardstick No.2 (*reviewing the requirements*): This yardstick investigates whether there was any review of the system requirements during the development process and the customer involvement. This review could be done by the developer in collaboration with the customer at the beginning of each iteration. Case 2 support this yardstick for graphical evidence. More precisely, the evidence is located in TABLE III. C-19-C USER ACCEPTANCE TEST RESULTS. The system requirements were reviewed by the customer through the acceptance test, meaning there was a participation of the customer during the development process.

Yardstick No.3 (*adapting to requirements changing*): This yardstick investigates whether there is any adaptation to a change in requirements during the development process, when the customer gives early feedback or new requirements. Case 2 support this yardstick with graphical evidence. More precisely, this evidence is located in TABLE III. C-19-C USER ACCEPTANCE TEST RESULTS, where the customer tests each completed iteration and gives early feedback.

Yardstick No.4 (*documentation for software requirements and specifications*): This yardstick investigates the existence of documentation for system requirements and specifications. Case 2 fully support this yardstick for graphical evidence, textual evidence and modelling evidence. This evidence is are located in TABLE I. C-19-C USE CASES, Section VI. B. Application implementation, and Figure 1. High-level use case diagram. These pieces of evidence contain descriptions of the system requirements to be implemented.

Yardstick No.5 (*documentation for source code*): This yardstick investigates the existence of source code documentation such as code modification, and reviews should be recorded to be submitted as evidence. Case 2 does not support this yardstick for coding evidence as there was no code evidence in the published case study. The authors mentioned in the section on programming languages that they used Flutter as their framework to develop their application, but evidence of the source code is required to support this yardstick.

Yardstick No.6 (*documentation for traceability purposes*): This yardstick investigates the existence of tractability documentation. The focus here is on documents that show the tractability of the system requirements and design phases. Case 2 partially support this yardstick with textual evidence located in Section B. User acceptance tests. More evidence is required to support this yardstick.

Yardstick No.7 (*identification of quality assurance characteristics*): The focus of this yardstick is on the existence of quality assurance characteristics. This yardstick investigates whether quality assurance characteristics such as functional suitability, performance efficiency, compatibility, usability, reliability, security, maintainability and portability were considered during the development process. Case 2 support this yardstick, with textual evidence and graphical evidence located in Section B. User acceptance tests and TABLE II. C-19-C TEST CASES.

Yardstick No.8 (*planning of risk management system*): This yardstick investigates whether there

is any planning for the risk management system to check whether system risks have been identified in the early developing phase. Case 2 supports this yardstick for textual evidence in Section V. REQUIREMENTS ELICITATION and Section VII. D. Limitations with plausible solutions.

Yardstick No.9 (*process of risk management system)*: This yardstick investigates the process of the risk management system to check that the risks have been detected, reviewed, and maintained continuously. Case 2 supports this yardstick for graphical evidence in TABLE II C19-C TEST CASES and TABLE III C-19-C USER ACCEPTANCE TEST RESULTS.

Yardstick No 10. (*verification*): This yardstick ensures that the system meets the intended requirements and the development process is on the right track. Case 2 fully supports this yardstick for graphical evidence in TABLE III. C-19-C USER ACCEPTANCE TEST RESULTS, Figure 2. Login screen, Figure 3. Signup screen, Figure 4. Phone verification Screen, Figure 5. NH user main screen, Figure 6. WH user main screen, Figure 7. Help request details - owner view, and Figure 8. Help request details-public view.

Yardstick No 11. (*validation*): This yardstick investigates that the system meets the required specifications and requirements. The validation process is usually accomplished by testing the software. Case 2 fully supports this yardstick for graphical evidence in TABLE II. C-19-C TEST CASES and TABLE III.C-19-C USER ACCEPTANCE TEST RESULTS. Auditing Decision and feedback from authors After applying the proposed auditing model to this case study, it has been found that this case study fully supports nine auditing yardsticks out of 11. One auditing yardstick is partially supported and the other one does not exist. Therefore, more evidence needs to be provided.

Case 2 Author's feedback

For this case study the authors of Case 2 were contacted, and were sent them the proposed auditing model and the auditing findings of their project. Their feedback was positive regarding the proposed auditing model and regarding the auditing findings they said "I do agree with your findings here as we did not include any source code in the paper. Thank you very much for pointing it out", and they were happy for their project to be audited. In the near future, we intend to work together to apply for MDR to get the CE mark.

6.4.3 Case 3

Title: A Web-Based, Mobile-Responsive Application to Screen Health Care Workers for COVID-19 Symptoms: Rapid Design, Deployment, and Usage [122].

Authors: Zhang et al.

Purpose: The aim of this study is to describe the design and the development process of the COVID Pass COVID-19 symptom screening application. The authors used agile principles to design, implement, and test their application and developed the minimum viable product of the mobile-responsive, web-based, and self-service application over the span of one week by using customised agile development process. This study was conducted at Partners HealthCare, a not-for-profit, academic, integrated health care delivery system in Boston, Massachusetts, United States.

Auditing findings

A summary of the auditing findings is shown in Table 6.3, with more details below.

Table 6.3: Summarised auditing findings for Case 3.

Case title: A Web-Based, Mobile-Responsive Application to Screen Health Care Workers for COVID-19 Symptoms: Rapid Design, Deployment, and Usage.				
Auditing Criteria and Yardsticks	**Evidence Existence (Yes/No/Partially)**	**Evidence Type**	**Evidence Location**	**Page Number**
Design Quality system procedure for Agile XP				
Yardstick No.1 (*planning for quality system procedure*)	Partially	Textual evidence	Section of Methods	2
Yardstick No.2 (*reviewing the requirements*)	Yes	Textual evidence	Section of Methods and Section of Discussion	5 and 7
Yardstick No.3 (*adapting to requirements changing*)	Yes	Textual evidence	Section of Discussion	7 and 8
Design software documentation management process for Agile XP				
Yardstick No.4 (*documentation for software requirements and specifications*)	Partially	Modelling evidence	Figure 4. Schematic of the process and procedures involved in using the COVID Pass application.	4
Yardstick No.5 *(documentation for source code)*	Yes	Codes evidence	Appendix1: Partners HealthCare. GitHub. URL: `https://github.com/partnershealthcare`	9
Yardstick No.6 (*documentation for tractability purposes*)	Partially	Textual evidence	Section of Discussion	8
Design production quality assurance for Agile XP				

Yardstick No.7 (*identification of quality assurance characteristics*)	Yes	Textual Evidence	Section of Methods Section of Discussion	5 8
Design risk management system for Agile XP				
Yardstick No.8 (*planning of risk management system*)	Partially	Textual evidence	Section of Limitations	8
Yardstick No.9 (*process of risk management system*)	No			
Design of verification and validation process for Agile XP				
Yardstick No.10 (*verification*)	Yes	Graphical evidence	Figure 1. Employee symptom reporting screen of the COVID Pass application. Figure 2. "Cleared for Work" screen of the COVID Pass application. Figure 3. "Not Cleared for Work" screen of the COVID Pass application.	3 3 4
Yardstick No.11 (*validation*)	Yes	Textual evidence and Graphical evidence	Section of Result Table 1. COVID Pass attestations by employee. Figure 7. Average daily employee attestations from Monday to Friday (COVID Pass, manual screening, and kiosk) by hour of day during Week 1 (March 30 to April 3, 2020) and Week 13 (June 22 to 26, 2020).	6 6 7

Yardstick No.1 (*planning for quality system procedure*): This yardstick investigates the existence

of the quality system procedure, looking for any evidence of planning for a quality system procedure such as a user story that contains the system requirements. Case 3 partially supports this yardstick for textual evidence in the section of methods, as there was no precise user story that contains the requirements of the system. A clear user story describing the requirements is required.

Yardstick No.2 (*reviewing the requirements*): This yardstick investigates whether there was any review of the system requirements during the development process and the customer involvement. This could be done by the developer in collaboration with the customer at the beginning of each iteration. Case 3 supports this yardstick for textual evidence. More precisely, this evidence is located in the methods section on page 5 and a section of the discussion on page 7. There was an involvement of the customer, who reviewed each iteration and gave early feedback.

Yardstick No.3 (*adapting to requirements changing*): This yardstick investigates whether there is any adaptation to changes in requirements during the development process, when the customer gives early feedback or new requirements. Case 3 supports this yardstick for textual evidence, which is located in a section of the discussion on pages 7 and 8 where the authors show that they have received early feedback with requirement changes after implementing the MVP of COVID pass, and they were able to create rapid changes and adjustments to the COVID Pass.

Yardstick No.4 (*documentation for software requirements and specifications*): This yardstick investigates the existence of the documentation for system requirements and specifications. Case 3 partially supports this yardstick for modelling evidence in Figure 4, a schematic of the process and procedures involved in using the COVID Pass application. This figure shows the class diagram for the requirements of the COVID Pass, but more evidence is required, such as a user story card and use case diagram with descriptions of the system requirements to be implemented.

Yardstick No.5 (*documentation for source code*): This yardstick investigates the existence of source code documentation such as code modification, and reviews should be recorded to be submitted as evidence. Case 3 supports this yardstick for coding evidence. In Appendix1: Partners HealthCare. GitHub. URL: `https://github.com/partnershealthcare` the authors have shared their source code to help any other institutions that may want to implement a similar solution.

Yardstick No.6 (*documentation for tractability purposes*): This yardstick investigates the existence of tractability documentation. The focus here is on documents that show the tractability of the system requirements and design phases. Case 3 partially supports this yardstick with textual evidence that is located in a section of the discussion, but more evidence is required to support this yardstick.

Yardstick No.7 (*identification of quality assurance characteristics*): The focus of this yardstick is on whether quality assurance characteristics such as functional suitability, performance efficiency, compatibility, usability, reliability, security, maintainability and portability were implemented during the development process. Case 3 supports this yardstick for textual evidence, which is located in section of methods on page 5 and section of discussion on page 8.

Yardstick No.8 (*planning of risk management system*): This yardstick investigates whether there

is any planning for the risk management system to check whether system risks have been identified in the early developing phase. Case 3 partially supports this yardstick for textual evidence in the section on limitations. More evidence is required to show the planning of the risk management system.

Yardstick No.9 (*process of risk management system*): This yardstick investigates the process of the risk management system to check that the risks have been detected, reviewed, and maintained continuously. Case 3 does not support this yardstick as there is no clear evidence showing that risks have been detected, reviewed, and maintained.

Yardstick No 10. (*verification*): This yardstick ensures that the system meets the intended requirements and the development process is on the right track. Case 3 fully support this yardstick for graphical evidence in Figure 1, Employee symptom reporting screen of the COVID Pass application, Figure 2, "Cleared for Work" screen of the COVID Pass application, and Figure 3, "Not Cleared for Work" screen of the COVID Pass application, as these figures show that the application is working as required.

Yardstick No 11. (*validation*): This yardstick investigates whether the system meets the required specifications and requirements. The validation process is usually accomplished by testing the software. Case 3 supports this yardstick for textual evidence and graphical evidence in results section, Table 1, COVID Pass attestations by employees, and Figure 7, Average daily employee attestations from Monday to Friday (COVID Pass, manual screening, and kiosk) by the hour of the day during Week 1 (March 30 to April 3, 2020) and Week 13 (June 22 to 26, 2020).

Auditing decision

After applying the proposed auditing model to this case study, it has been found that this case study fully supports six auditing yardsticks out of 11. Four auditing yardsticks are partially supported and the other one does not exist. Therefore more evidence is required.

6.4.4 Case 4

Title: Agile Software Development with Open Source Software in a Hospital Environment – Case Study of an eCRF-System for Orthopaedical Studies [123]

Authors: T¨unay Ozcan, Semra Kocak, and Philipp Brune.

Purpose: The aim of the research is to show how to successfully develop a web-based application in a clinical environment using a tailored agile methodology. The authors developed an electronic case report form (eCRF) application for orthopaedical studies. This study took place in Ulm University Hospital, Germany. Due to the close interaction with the relevant stakeholders (in particular physicians) and the unclear detail requirements at the project start, an agile method was chosen for the project.

Auditing findings

A summary of the auditing findings is shown in Table 6.4, with more details below.

Table 6.4: Summarised auditing findings for Case 4.

Case title: Agile Software Development with Open Source Software in a Hospital Environment Case Study of an eCRF-System for Orthopaedical Studies				
Auditing Criteria and Yardsticks	**Evidence Existence (Yes/No/Partially)**	**Evidence Type**	**Evidence Location**	**Page Number**
Design Quality system procedure for Agile XP				
Yardstick No.1 (*planning for quality system procedure*)	Yes	Textual evidence	Section 3. Project Context Section 4. Agile Methodology	442 443 and 445
Yardstick No.2 (*reviewing the requirements*)	Yes	Textual evidence	Section 4. Agile Methodology	444
Yardstick No.3 (*adapting to requirements changing*)	Yes	Textual evidence And Graphical evidence	Section 4. Agile Methodology. Fig. 4. Snapshot of the information board used for communication between the team members.	446
Design software documentation management process for Agile XP				
Yardstick No.4 (*documentation for software requirements and specifications*)	Partially	Graphical evidence and Textual evidence	Fig. 2. Functional modules of OrthoClinical. Section 4. Agile Methodology	444 446
Yardstick No.5 *(documentation for source code)*	No			
Yardstick No.6 (*documentation for tractability purposes*)	Partially	Textual evidence	Section 4. Agile Methodology	446
Design production quality assurance for Agile XP				
Yardstick No.7 (*identification of quality assurance characteristics*)	Partially	Textual evidence	Section 4. Agile Methodology Section 6. Prototype Evaluation	446 448

Design risk management system for Agile XP				
Yardstick No.8 (*planning of risk management system*)	No			
Yardstick No.9 (*process of risk management system*)	No			
Design of verification and validation process for Agile XP				
Yardstick No.10 (*verification*)	Partially	Textual evidence and Graphical evidence	Section 6. Prototype Evaluation. Table 1. Results of the usability test of OrthoClinical from the patients' perspective	449
Yardstick No.11 (*validation*)	Yes	Graphical evidence and Textual evidence	Fig. 5. Screenshot of a questionnaire form. Section 6. Prototype Evaluation	448

Yardstick No.1 (*planning for quality system procedure*): This yardstick investigates the quality system procedure by looking for any evidence of planning for a quality system procedure, such as a user story that contains the system requirements. Case 4 supports this yardstick for textual evidence. More precisely, the evidence is located in Section 3, Project Context page 442 and Section 4, Agile Methodology pages 443 and 445. This evidence describes the exploration phase of the system development and the gathering of the requirements from the people involved in this study and estimates the expected value.

Yardstick No.2 (*reviewing the requirements*): This yardstick investigates whether there was any review of the system requirements during the development process and customer involvement. The review of the requirements could be done by the developer in collaboration with the customer at the beginning of each iteration. Case 4 support this yardstick for textual evidence. More precisely, the evidence is located in Section 4, Agile Methodology page 444, where the authors state there was a daily meeting to review requirements and to keep up with the latest iteration. This meeting was attended by an expert user, lead designer, designer-programmer and coordinator, meaning the user was involved during the development process.

Yardstick No.3 (*adapting to requirements changing*): This yardstick investigates whether there is any adaptation to requirements changes during the development process when the customer gives early feedback or new requirements. Case 4 supports this yardstick for textual evidence and graphical evidence. More precisely, this evidence is located in Section 4, Agile Methodology page 446 and Fig. 4, Snapshot of

the information board used for communication between the team members, where the expert user gives early feedback on the user story, acceptance criteria, and prioritising.

Yardstick No.4 (*documentation for software requirements and specifications*): This yardstick investigates the existence of documentation for system requirements and specifications. Case 4 partially support this yardstick for graphical evidence and textual evidence. This evidence is located in Fig. 2, Functional modules of OrthoClinical and Section 4, Agile Methodology page 444. There are no documents for the actual user story, therefore more evidence is required.

Yardstick No.5 (*documentation for source code*): This yardstick investigates the existence of source code documentation such as code modification, and any reviews should be recorded to be submitted as evidence. Case 4 does not support this yardstick for coding evidence. The authors mentioned that they used Java EE and PHP, but there was no code evidence in the published case study. Evidence of the source code is required in order to support this yardstick.

Yardstick No.6 (*documentation for tractability purposes*): This yardstick investigates the existence of tractability documentation. The focus here is on documents that show the tractability of the system requirements and design phases. Case 4 partially support this yardstick, with textual evidence located in Section 4, Agile Methodology page 446. There is no clear evidence showing the documentation for tractability purposes. Therefore, more evidence is required.

Yardstick No.7 (*identification of quality assurance characteristics*): The focus of this yardstick is on the existence of quality assurance characteristics. This yardstick investigates whether quality assurance characteristics such as functional suitability, performance efficiency, compatibility, usability, reliability, security, maintainability and portability were implemented during the development process. Case 4 partially support this yardstick for textual evidence located in Section 4, Agile Methodology and Section 6, Prototype Evaluation. Clearer evidence is required.

Yardstick No.8 (*planning of risk management system*): This yardstick investigates whether there is any planning for the risk management system, to check whether system risks were identified in the early development phase. Case 4 does not support this yardstick. Evidence of the planning of a risk management system is required in order to support this yardstick.

Yardstick No.9 (*process of risk management system*): This yardstick investigates the process of the risk management system to check that the risks have been detected, reviewed, and maintained continuously. Case 4 does not support this yardstick. Evidence of the process of a risk management system is required in order to support this yardstick.

Yardstick No 10. (*Verification*): This yardstick ensures that the system meets the intended requirements and the development process is on the right track. Case 4 partially support this yardstick for textual evidence and graphical evidence in Section 6, Prototype Evaluation and Table 1, Results of the usability test of OrthoClinical from the patients' perspective.

Yardstick No 11. (*Validation*): This yardstick investigates whether the system meets the required specifications and requirements. The validation process is usually accomplished by testing the software.

Case 4 supports this yardstick for graphical evidence and textual evidence in Fig. 5, Screenshot of a questionnaire form and Section 6, Prototype Evaluation.

Auditing decision

After applying the proposed auditing model to this case study, it has been found that this case study fully supports four auditing yardsticks out of 11. Four auditing yardsticks are partially supported and three auditing yardsticks do not exist. Therefore, more evidence is required.

6.4.5 Case 5

Title: Agile methods for open source safety-critical software [124]

Authors: Kevin Gary et.al.

Purpose: This research paper describes how to adopt agile methods to develop safety-critical software. The authors adopted an agile methodology and tailored it to their needs to develop an image-guided surgical toolkit (IGSTK). IGSTK is an open source project that relies on the collaboration of a skilled distributed development team to construct a cross-platform application framework in a safety-critical domain [124]. IGSTK has been used in teaching hospitals, research labs, and used for clinical trials and also has received a determination of non-significant risk from the FDA.

Auditing findings

A summary of the auditing findings is shown in Table 6.5, with more details below.

Table 6.5: Summarised auditing findings for Case 5.

Case title: Agile methods for open source safety-critical software				
Auditing Criteria and Yardsticks	**Evidence Existence (Yes/No/Partially)**	**Evidence Type**	**Evidence Location**	**Page Number**
Design Quality system procedure for Agile XP				
Yardstick No.1 (*planning for quality system procedure*)	Partially	Textual evidence	Section 4.2. Requirements management.	8
Yardstick No.2 (*reviewing the requirements*)	Yes	Textual evidence and modelling evidence	Section 4.2. Requirements management. Figure 4. Requirements Management process in IGSTK.	8 9

Yardstick No.3 (*adapting to requirements changing*)	Yes	Textual evidence	Section 4.2. Requirements management.	8
Design software documentation management process for Agile XP				
Yardstick No.4 (*documentation for software requirements and specifications*)	Partially	Modelling evidence	Figure 5. IGSTK components and connectors. APPENDIX: ARCHITECTURAL APPROACH DESCRIPTIONS	9 15
Yardstick No.5 *(documentation for source code)*	No			
Yardstick No.6 (*documentation for tractability purposes*)	No			
Design production quality assurance for Agile XP				
Yardstick No.7 (*identification of quality assurance characteristics*)	Yes	Graphical evidence	Table I. Quality Attribute Utility Tree for IGSTK.	13
Design risk management system for Agile XP				
Yardstick No.8 (*planning of risk management system*)	Yes	Textual evidence	4.4. Continuous integration and testing	10
Yardstick No.9 (*process of risk management system*)	Yes	Textual evidence	4.3. Safety by design 4.4. Continuous integration and testing.	8 10
Design of verification and validation process for Agile XP				
Yardstick No.10 (*verification*)	Yes	Textual evidence	4.1. Best practices	7

Yardstick No.11 (*validation*)	Yes	Textual evidence and Graphical evidence	4.5. Agile architecture validation in IGSTK. Figure 7. CDash dashboard displaying IGSTK nightly build and test results.	11 11

Yardstick No.1 (*planning for quality system procedure*): This yardstick investigates the existence of the quality system procedure by looking for any evidence of planning for quality system procedures such as a user story that contains the system requirements. Case 5 partially supports this yardstick for textual evidence. More precisely, this evidence is located in Section 4.2. Requirements management. The authors mentioned that all verified requirements are automatically extracted into Latex and PDF files and archived. However, in this research paper, there is no clear user story or requirements of the system mentioned or illustrated. Therefore, more evidence is required to fully support this yardstick.

Yardstick No.2 (*reviewing the requirements*): This yardstick investigates whether there was any review of the system requirements during the development process and the customer involvement. The review of the requirements should be done by the developer in collaboration with the customer at the beginning of each iteration. Case 5 supports this yardstick for textual evidence and modelling evidence. More precisely, this evidence is located in Section 4.2, Requirements management and Figure 4, Requirements management process in IGSTK.

Yardstick No.3 (*adapting to requirements changing*): This yardstick investigates whether there is any adaptation to requirement changes during the development process, when the customer gives early feedback or new requirements. Case 5 support this yardstick for textual evidence. More precisely, this evidence is located in Section 4.2, Requirements management.

Yardstick No.4 (*documentation for software requirements and specifications*): This yardstick investigate the existence of the documentation for system requirements and specification. Case 5 partially supports this yardstick for modelling evidence. This evidence is located in Figure 5, IGSTK components and connectors, and appendix: architectural approach descriptions. The authors mentioned that they have a wiki and auto pdf for user stories, but in this research paper there is no clear documentation illustrating the system requirements. Therefore, more evidence such as a user story and/or UML are required to fully support this yardstick.

Yardstick No.5 (*documentation for source code*): This yardstick investigates the existence of source code documentation such as code modification, and reviews should be recorded to be submitted as evidence. Case 5 does not support this yardstick for coding evidence. The authors mentioned that they used Use sandboxes for evolving code, but no codes were illustrated in this research paper. Evidence of the source code is required in order to support this yardstick.

Yardstick No.6 (*documentation for tractability purposes*): This yardstick investigates the exis-

tence of tractability documentation. The focus here is on documents that show the tractability of the system requirements and design phases. Case 5 does not support this yardstick. The authors mentioned that the system requirements and documentation are archived in Wiki log so that they can be reopened later for tractability purposes. But in this research paper, there is no evidence showing that documentation for traceability purposes. Therefore, more evidence is required.

Yardstick No.7 (*identification of quality assurance characteristics*): The focus of this yardstick is on the existence of quality assurance characteristics. This yardstick investigates whether quality assurance characteristics such as functional suitability, performance efficiency, compatibility, usability, reliability, security, maintainability and portability were implemented during the development process. Case 5 supports this yardstick for graphical evidence, which is located in Table I, Quality Attribute Utility Tree for IGSTK.

Yardstick No.8 (*planning of risk management system*): This yardstick investigates whether there is any planning for the risk management system to check whether system risks were identified in the early development phase. Case 5 supports this yardstick for textual evidence. More precisely, this evidence is located in Section 4.4, Continuous integration and testing.

Yardstick No.9 (*process of risk management system*): this yardstick investigates the process of the risk management system to check that the risks have been detected, reviewed, and maintained continuously. Case 5 supports this yardstick for textual evidence. This evidence is located in Section 4.3, Safety by design, and Section 4.4, Continuous integration and testing.

Yardstick No 10. (*verification*): This yardstick ensures that the system meets the intended requirements and the development process is on the right track. Case 5 supports this yardstick for textual evidence in Section 4.1, Best practices.

Yardstick No 11. (*validation*): This yardstick investigates whether the system meets the required specifications and requirements. The validation process is usually accomplished by testing the software. Case 5 supports this yardstick for textual evidence and graphical evidence. This evidence is located in Section 4.5, Agile architecture validation in IGSTK and Figure 7, CDash dashboard displaying IGSTK nightly build and test results.

Auditing decision.

After applying the proposed auditing model to this case study, it has been found that t this case study fully supports seven auditing yardsticks out of 11. Two auditing yardsticks are partially supported and two auditing yardsticks do not exist. Therefore, more evidence is required.

6.4.6 Case 6

Title: On designing a usable interactive system to support transplant nursing [125]

Authors: A. Narasimhadevara, T. Radhakrishnan, B. Leung, and R. Jayakumar.

Purpose: the aim of this research is to combine two well-known software engineering techniques—namely,

agile programming and user-centred design—to develop an interactive system for supporting the activities of transplant nurses in a hospital setting. The software product developed has been well accepted and is currently planned to replace the manual methods followed in the transplant ward of a large metropolitan hospital. **Auditing findings**

A summary of the auditing findings is shown in Table 6.6, with more details below.

Table 6.6: Summarised auditing findings for Case 6.

On designing a usable interactive system to support transplant nursing				
Auditing Criteria and Yardsticks	**Evidence Existence (Yes/No/Partially)**	**Evidence Type**	**Evidence Location**	**Page Number**
Design Quality system procedure for Agile XP				
Yardstick No.1 (*planning for quality system procedure*)	Yes	Textual evidence and Modelling evidence	Section 3. Understanding the users, tasks and environment. Fig. 2. The eight processes in a transplant ward. 5.1. NUI design progress, Stage 1.	3 4 6
Yardstick No.2 (*reviewing the requirements*)	Partially	Textual evidence	Section 4. Agile method and user-centred design: 1. Developing software in short iterations.	5
Yardstick No.3 (*adapting to requirements changing*)	Yes	Textual evidence	Section 4. Agile method and user-centred design: 2.Interactive communication. Section 5.2. NUI design progress, Stage 2.	5 7
Design software documentation management process for Agile XP				

Yardstick No.4 (*documentation for software requirements and specifications*)	Yes	Textual evidence and Modelling evidence	Appendix A. Supplementary data. Fig. 2. The eight processes in a transplant ward. Fig. 3. NUI design process, Stage 1. Fig. 5. NUI design process, Stage 3.	15 4 6 7
Yardstick No.5 *(documentation for source code)*	No			
Yardstick No.6 (*documentation for tractability purposes*)	Partially	Textual evidence	Section 4. Agile method and user-centred design: 1. Developing software in short iterations. 5.2. NUI design progress, Stage 2 for	5 7
Design production quality assurance for Agile XP				
Yardstick No.7 (*identification of quality assurance characteristics*)	Yes	Textual evidence	Section 5.2. NUI design progress, Stage 2. Section 5.3. NUI design progress, Stage 3	7 8
Design risk management system for Agile XP				
Yardstick No.8 (*planning of risk management system*)	Yes	Textual evidence and Graphical evidence	Section 5.3. NUI design progress, Stage 3. Table 1, Steps followed in NUI development.	8
Yardstick No.9 (*process of risk management system*)	Yes	Textual evidence And Graphical evidence	Section 5.3. NUI design progress, Stage 3. Table 1Steps followed in NUI development.	8
Design of verification and validation process for Agile XP				

Yardstick No.10 (*verification*)	Yes	Textual evidence and Graphical evidence	Section 6. The user interface of NUI. Fig. 6. NUI overview panel. Fig. 8. NUI medication panel. Fig. 9. Glucose panel. Fig. 10. Glucose report graph. Fig. 11. Progression of the NUI pilot study.	8 9 11 12 13
Yardstick No.11 (*validation*)	Yes	Textual evidence	Section 7. Planning and conducting the field test Section 8. Usability evaluation	10 13

Yardstick No.1 (*planning for quality system procedure*): This yardstick investigates the existence of the quality system procedure, looking for any evidence of planning for quality system procedures such as a user story that contains the system requirements. Case 6 supports this yardstick for textual evidence and modelling evidence. More precisely, this evidence is located in Section 3, Understanding the users, tasks and environment, 5.1, NUI design progress, Stage 1, and Fig. 2, the eight processes in a transplant ward. This evidence describes the exploration phase of the system development.

Yardstick No.2 (*reviewing the requirements*): This yardstick investigates whether there was any review of the system requirements during the development process and the customer involvement. The review of the requirements could be done by the developer in collaboration with the customer at the beginning of each iteration. Case 6 partially supports this yardstick for textual evidence. This evidenceis located in Section 4, Agile method and User-Centred Design: 1. Developing software in short iterations. There is no clear description showing how the developer and customer reviewed the system requirements. More evidence is required in order to fully support this yardstick.

Yardstick No.3 (*adapting to requirements changing*): This yardstick investigates whether there is any adaptation to the requirements or changes during the development process when the customer gives early feedback or new requirements. Case 6 supports this yardstick for textual evidence. More precisely, this evidence is located in Section 4, Agile method and User-Centred Design: 2. Interactive communication and Section 5.2. NUI design progress, Stage 2, where there was real-time face-to-face communication between customers and developers during the development process and also there was an acceptance test where users test the system and give early feedback.

Yardstick No.4 (*documentation for software requirements and specifications*): This yardstick investigate the existence of the documentation for system requirements and specification. Case 6 fully support this yardstick for textual evidence and modelling evidence. This evidence is located in Appendix A, Supplementary data, Fig. 2, The eight processes in a transplant ward, Fig. 3, NUI design process,

Stage 1, and Fig. 5, NUI design process, Stage 3. This evidence contains descriptions of the system requirements and specifications to be implemented.

Yardstick No.5 (*documentation for source code*): this yardstick investigate the existence of the source code documentation such as code modification and review should be recorded to be submitted as evidence. Case 6 does not support this yardstick for coding evidence. As there was no codes evidence in the published case study. Evidence of the source code is required in order to support this yardstick.

Yardstick No.6 (*documentation for tractability purposes*): This yardstick investigates the existence of the tractability of documentation. The focus here is on documents that show the tractability of the system requirements and design phases. Case 6 partially support this yardstick with textual evidence. This evidence is located in Section 4, Agile method and User-Centred Design: 1. Developing software in short iterations and 5.2. NUI design progress, Stage 2. More evidence is required to fully support this yardstick.

Yardstick No.7 (*identification of quality assurance characteristics*): The focus of this yardstick is on the existence of the quality assurance characteristics. This yardstick investigates whether the quality assurance characteristics such as functional suitability, performance efficiency, compatibility, usability, reliability, security, maintainability and portability were implemented during the development process. Case 6 supports this yardstick for textual evidence. This evidence is located in Section 5.2. NUI design progress, Stage 2 and Section 5.3. NUI design progress, Stage 3.

Yardstick No.8 (*planning of risk management system*): This yardstick investigates whether there is any planning for the risk management system to check whether system risks have been identified in the early development phase. Case 6 supports this yardstick for textual evidence and graphical evidence in Section 5.3, NUI design progress, Stag -3 and Table 1, Steps followed in NUI development.

Yardstick No.9 (*process of risk management system*): This yardstick investigates the process of the risk management system to check that the risks have been detected, reviewed, and maintained continuously. Case 6 supports this yardstick for textual evidence and graphical evidence in Section 5.3, NUI design progress, Stage 3 and Table 1, Steps followed in NUI development.

Yardstick No 10. (*verification*): This yardstick ensures that the system meets the intended requirements and the development process is on the right track. Case 6 fully support this yardstick for textual evidence and graphical evidence in Section 6, the user interface of NUI, Fig. 6. NUI overview panel, Fig. 8. NUI medication panel, Fig. 9. Glucose panel, Fig. 10. Glucose report graph, and Fig. 11. Progression of the NUI pilot study.

Yardstick No 11. (*validation*): This yardstick investigates that the system meets the required specifications and requirements. The validation process is usually accomplished by testing the software. Case 6 fully supports this yardstick for textual evidence in Section 7, Planning and conducting the field test and Section 8, Usability evaluation.

Auditing decision

After applying the proposed auditing model to this case study, it has been found that this case study fully supports eight auditing yardsticks out of 11. One auditing yardstick is partially supported and another one does not exist. Therefore more evidence is required.

6.5 Case 7

Title: Development of Medical Records with Extreme Programming SDLC [126]

Authors: I Gusti Ngurah Suryantara and Johanes Fernandes Andry

Purpose: the aim of this project is to develop medical record applications using Agile XP as their primary SDLC.

Auditing findings

A summary of the auditing findings is shown in Table 6.7, with more details below.

Table 6.7: Summarised auditing findings for Case 7.

Case title: Development of Medical Record with Extreme Programming SDLC				
Auditing Criteria and Yardsticks	**Evidence Existence (Yes/No/Partially)**	**Evidence Type**	**Evidence Location**	**Page Number**
Design Quality system procedure for Agile XP				
Yardstick No.1 (*planning for quality system procedure*)	Yes	Textual evidence and Graphical evidence	IV.A. Requirement. IV.E. Planning. Fig 4. Step by Step of Research Methods. Table 2. User Stories	4 5
Yardstick No.2 (*reviewing the requirements*)	Partially	Textual evidence and Graphical evidence	IV.D. Schedule Development Fig 6. Life Cycle and Schedule Development	5
Yardstick No.3 (*adapting to requirements changing*)	Yes	Textual evidence	I. Software Increment	7
Design software documentation management process for Agile XP				

Yardstick No.4 (*documentation for software requirements and specifications*)	Yes	Graphical evidence	Fig 4. Step by Step of Research Methods. Fig 5. Cycle of Applications Table 2. User Stories Fig 9. Documentation Table.	4 5 6
Yardstick No.5 *(documentation for source code)*	Partially	Graphical evidence	Table 5. Mapping of tables in database. Table 6. Mapping user interface Fig 10. Documentation of Coding	6
Yardstick No.6 (*documentation for tractability purposes*)	Yes	Textual evidence and Graphical evidence	IV.F. Design Table 4. Design CRC and Mapping of Class. Fig 7. Mapping of CRC	5
Design production quality assurance for Agile XP				
Yardstick No.7 (*identification of quality assurance characteristics*)	Partially	Textual evidence	IV.H. Testing	6
Design risk management system for Agile XP				
Yardstick No.8 (*planning of risk management system*)	Yes	Textual evidence and Graphical evidence	IV.H. Testing Table 7. Mapping result of testing with white box. Table 8. Mapping result testing with black box	6 7
Yardstick No.9 (*process of risk management system*)	No			
Design of verification and validation process for Agile XP				
Yardstick No.10 (*verification*)	Yes	Textual evidence and Graphical evidence	IV.H. Testing Table 7. Mapping result of testing with white box. Table 8. Mapping result testing with black box	6 7

Yardstick No.11 (*validation*)	Yes	Textual evidence and Graphical evidence	IV.H. Testing Table 7. Mapping result of testing with white box. Table 8. Mapping result testing with black box	6 7

Yardstick No.1 (*planning for quality system procedure*): this yardstick investigates the existence of the quality system procedure, looking for any evidence of planning for quality system procedures, such as a user story that contains the system requirements. Case 7 supports this yardstick for textual evidence and graphical evidence. More precisely, this evidence is located in Section IV. A. Requirement, IV. E. Planning, Fig 4. Step by Step of Research Methods, and Table 2. User Stories.

Yardstick No.2 (*reviewing the requirements*): This yardstick investigates whether there was any review of the system requirements during the development process and the customer involvement. The review of the requirements could be done by the developer in collaboration with the customer at the beginning of each iteration. Case 7 partially support this yardstick for textual evidence and graphical evidence. More precisely, the evidence is located in Section IV. D. Schedule Development and Fig 6. Life Cycle and Schedule Development. As there is no obvious evidence showing that the requirements were reviewed at the beginning of each iteration in collaboration with the customer, more evidence is required in order to fully support this yardstick.

Yardstick No.3 (*adapting to requirements changing*): This yardstick investigates whether there is any adaptation to the requirements or changes during the development process, when customers give early feedback or new requirements. Case 7 supports this yardstick for textual evidence. More precisely, this evidence is located in Section I, Software Increment.

Yardstick No.4 (*documentation for software requirements and specifications*): This yardstick investigates the existence of the documentation for system requirements and specifications. Case 7 supports this yardstick for graphical evidence. This evidence is located in Fig 4. Step by Step of Research Methods, Fig 5. Cycle of Applications, Table 2. User Stories, and Fig 9. Documentation Table.

Yardstick No.5 (*documentation for source code*): This yardstick investigates the existence of source code documentation such as code modification, and reviews should be recorded to be submitted as evidence. Case 7 partially supports this yardstick for graphical evidence. This evidence is illustrated in Table 5, Mapping of tables in databases, Table 6, Mapping user interface, and Fig 10, Documentation of Coding. However, the authors did not mention anything regarding the source code or comments on code. Evidence of the source code is required in order to fully support this yardstick.

Yardstick No.6 (*documentation for tractability purposes*): This yardstick investigates the existence of the tractability documentation. The focus here is on documents that show the tractability of the system requirements and design phases. Case 5 support this yardstick for textual evidence and graphical

evidence. This evidence is illustrated by Section IV. F. Design, Table 4. Design CRC and Mapping of Class, and Fig 7. Mapping of CRC.

Yardstick No.7 (*identification of quality assurance characteristics*): The focus of this yardstick is on the existence of the quality assurance characteristics. This yardstick investigates whether the quality assurance characteristics such as functional suitability, performance efficiency, compatibility, usability, reliability, security, maintainability and portability were implemented during the development process. Case 7 partially support this yardstick for textual evidence. This evidence is located in IV. H. Testing. More evidence is required to show that the developed software is either fault-free, or has a minimum number of faults.

Yardstick No.8 (*planning of risk management system*): This yardstick investigates whether there is any planning for the risk management system to check whether system risks have been identified in the early developing phase. Case 7 supports this yardstick for textual evidence and graphical evidence. More precisely, this evidence is located in Section IV. H. Testing, Table 7. Mapping result of testing with white box, Table 8. Mapping result testing with black box.

Yardstick No.9 (*process of risk management system*): This yardstick investigates the process of the risk management system to check that the risks have been detected, reviewed, and maintained continuously. Case 7 does not support this yardstick as there is no obvious evidence showing that the process of risk management system has been applied during the development, such as pair programming practice and refactoring practice. Therefore, more evidence is required to support this yardstick.

Yardstick No 10. (*verification*): This yardstick ensures that the system meets the intended requirements and the development process is on the right track. Case 7 supports this yardstick for textual evidence and graphical evidence. This evidence is illustrated by Section IV. H. Testing, Table 7. Mapping result of testing with white box, and Table 8. Mapping result testing with black box.

Yardstick No 11. (*validation*): This yardstick investigates that the system meets the required specifications and requirements. The validation process is usually accomplished by testing the software. Case 7 supports this yardstick for textual evidence and graphical evidence. This evidence is illustrated by Section IV. H. Testing, Table 7. Mapping result of testing with white box, and Table 8. Mapping result testing with black box.

Auditing decision

After applying the proposed auditing model to this case study, it has been found that this case study fully supports seven auditing yardsticks out of 11. Three auditing yardsticks are partially supported and one auditing yardstick does not exist. Therefore, more evidence is required.

6.6 Discussion and conclusion

Seven different case studies were audited based on the proposed auditing model to investigate whether or not they comply with the EU MDR requirements. Based on the auditing results of the selected case

studies (see Table 6.8 Evidence summary of the selected case studies), no case study has demonstrated full support to the auditing yardsticks of the proposed auditing model. Each case study would require more evidence in order to fully support the yardsticks and criteria that lead to the compliance with EU MDR requirements. However, the evidence gathered from each case study shows at least partial support for the auditing yardsticks.

We can notice that the yardsticks that are missing most frequently in the seven case studies are the ones regarding the design software documentation management process for Agile XP. For instance, yardstick No. 5, documentation for source code, is the one most frequently missing in the seven case studies. This is because the Agile XP process has light documentation for the development process. We also noticed that using the modified Agile XP process to develop medical device software would enable the developer to provide more evidence for compliance purposes. For example, Case 2, Case 5, and Case 7 used Agile XP with some modification to keep the agile as light as possible, and they supported approximately 80% of the yardsticks as fully supported and partially supported.

Therefore, using the proposed auditing model during the development process would enable the developer to modify the Agile XP practices in a way that will assist in preparing the required evidence to be submitted for EU MDR auditing. For example, the authors of Case 1 and Case 2, after reading the auditing results, confirmed that they should add the missing evidence to their case study. Moreover, the proposed auditing model would enable the auditor to easily extract the required evidence by following the yardsticks of the proposed auditing model.

6.7 Summary

This chapter explained the evaluation procedure of the proposed auditing model and the auditing results of the selected case studies. The evaluation was performed based on planning, examination, decision-making (PED) evaluation framework. Seven different case studies were audited based on the proposed auditing model to investigate whether or not they comply with the EU MDR requirements. The evidence gathered shows at least partial support for the requirements in each case study. However, no case study has been demonstrated as supporting fully the auditing yardsticks of the proposed auditing model.

Table 6.8: Evidence summary of the selected 7 case studies.

	Case 1	**Case 2**	**Case 3**	**Case 4**	**Case 5**	**Case 6**	**Case 7**
Design quality system procedure for Agile XP							
Yardstick No.1 (*planning for quality system procedure*)	Yes	Yes	Partially	Yes	Partially	Yes	Yes
Yardstick No.2 (*reviewing the requirements*)	Partially	Yes	Yes	Yes	Yes	Partially	Partially
Yardstick No.3 (*adapting to requirements changing*)	Partially	Yes	Yes	Yes	Yes	Yes	Yes
Design software documentation management process for Agile XP							
Yardstick No.4 (*documentation for software requirements and specifications*)	Yes	Yes	Partially	Partially	Partially	Yes	Yes
Yardstick No.5 *(documentation for source code)*	Yes	No	Yes	No	No	No	Partially
Yardstick No.6 (*documentation for tractability purposes*)	Yes	Partially	Partially	Partially	No	Partially	Yes
Design production quality assurance for Agile XP							
Yardstick No.7 (*identification of quality assurance characteristics*)	Yes	Yes	Yes	Partially	Yes	Yes	Partially
Design risk management system for Agile XP							
Yardstick No.8 (*planning of risk management system*)	Yes	Yes	Partially	No	Yes	Yes	Yes
Yardstick No.9 (*process of risk management system*)	Yes	Yes	No	No	Yes	Yes	No
Design of verification and validation process for Agile XP							
Yardstick No.10 (*verification*)	Yes	Yes	Yes	Partially	Yes	Yes	Yes
Yardstick No.11 (*validation*)	Yes	Yes	Yes	Yes	Yes	Yes	Yes

Chapter 7

Discussion and Future Work

The goal of this research project was to enhance the Agile XP practices in supporting the auditing requirements of EU MDR, and to help medical device software development companies that wish to use Agile XP practices to meet the EU MDR certification requirements (CE marking). This goal was reached through the achievement of two main objectives: first, proposing an extension to the Agile XP user story to enhance the early planning activities of Agile XP according to the EU MDR requirements. Second, designing an auditing model that covers the requirements of the EU MDR. This auditing model should provide the EU MDR auditors with auditing evidence that the medical device software developed with an Agile XP process has fulfilled the requirements of the EU MDR. This chapter will present a discussion of the main contributions of this thesis will make recommendations for future work.

7.1 Discussion

This research has implications for the medical device software development field as it has provided benefits that will enhance the use of Agile XP in developing medical device software. These benefits are summarised as follows:

- This research has minimised the ambiguities of the EU MDR requirements that impact directly or indirectly on the SDLC.
- The documentation of the Agile XP practices has been enhanced when developing medical device software.
- This research has added a formality to the Agile XP user story in a way that contains EU MDR requirements for the planning phase, such as functional requirements and non-functional requirements.
- This research has enhanced the Agile XP process in supporting the auditing requirements of the EU MDR regulations, which will help the Agile XP software organisations in their efforts to become CE marking certified and will help Agile XP developers to follow the EU MDR requirements more closely.

This research project will benefit people who are working on agile process improvement and software regulatory compliance, such as academic researchers who have an interest in enhancing the agile software development process within the regulatory environment, software developers who wish to use the Agile XP auditing model to develop medical device software in compliance with the EU MDR requirements, and information system auditors of EU MDR who need to extract the auditing evidence from Agile XP practices to ensure that the process of developing medical device software conforms to the EU MDR requirements.

Below is a description of the main contributions of the research that provided the aforementioned benefits.

Analysing of the agile software processes in the healthcare industry

This research analysed the characteristics of the most common agile processes and compared them based on the key features of a software development project, such as time, cost, flexibility, and documentation (more details can be found in Section 2.1.1: Traditional SDLC Models). The research also investigated the challenges faced by medical device software development companies when using an agile process. It has been found that there is a low adoption rate of agile practices in the healthcare industry and this is due to the challenges that medical device software developers face when using agile practices as their development methodology. These challenges include the lack of the following aspects: fixed up-front planning, documentation, traceability, and formality (for more details, see Section 2.3: Agile Practices in the Healthcare Industry).

Moreover, this research has investigated the suitability of the agile practices XP, Scrum, and FDD in the healthcare industry, in supporting the EU MDRs in particular. The investigation found that none of the agile practices is suitable for use in the development of medical device software (for more details, see Section 2.4: Suitability of Agile Practices in the Healthcare Industry).

Extraction of EU MDR requirements

This research has extracted the EU MDR requirements that have a direct or indirect impact on the SDLC when applying for an EU MDR compliance certificate (more details can be found in Appendix A). After the requirements were extracted, an analysis was conducted to detect the ambiguities of the EU MDR requirements. This research has proposed a solution to minimise the ambiguities in EU MDR requirements. This solution is based on software engineering techniques such as user story cards and UML diagrams (more details can be found in Section 3.6: Detecting the Ambiguities of the EU MDR Requirements).

Extension to the Agile XP user story

This research proposed extensions to the Agile XP user story to improve the planning phase of Agile XP. The proposed extension was based on five sub-processes derived from the ASM ground model and SPICE, such as ISO/IEC 25030:2019 and ISO/IEC 25010:2011. These sub-processes include identifying user story suppliers, ensuring the formality of functional requirements, proposing a semi-structured form for non-functional requirements, prioritisation of the user story, and identification of user story dependency. These sub-processes are integrated into the Agile XP user story to enhance its ability to collect and represent the

system requirements corresponding to the EU MDR requirements that are mandatory for the CE marking certificate (more details can be found in Section 4.5: The Extended Agile XP User Story). A mapping study was conducted to check the capability of the extended Agile XP user story in meeting the EU MDR requirements for the planning phase. As result, most of the EU MDR requirements for the planning phase were found in the proposed extension of the Agile XP user story (more details can be found in Section 4.6: Mapping between the Agile XP Extended User Story and the EU MDR). The extended Agile XP user story has several advantages, such as formality, the consideration of non-functional requirements, and the precise capturing of the functional requirements. In addition, risk management, traceability, and test reports are considered in the extended user story. Therefore, the extended Agile XP user story could be considered as a solution for up-front planning when developing medical device software.

An Auditing Model for EU MDR Requirements in the Agile XP Environment

This research has proposed an auditing model for the EU MDR requirements that is applicable in Agile XP software process environments. The proposed auditing model aims to enhance the auditability of Agile XP concerning the EU MDR requirements. The design of the auditing model is based on the framework of the evaluation method developed by Scriven. The proposed auditing model consists of four common evaluation components: identification of the target, design audit criteria and yardsticks, evidence gathering and synthesis, and auditing decisions (more details can be found in Section 5.2: Design Methodology of the Auditing Model). Moreover, the design of the auditing criteria and yardsticks could not be achieved without the support of quality management system standards such as BS ISO-IEC 12207, BS EN ISO 13485, BS ISO-IEC 25010, and BS EN ISO 14971. The criteria of the proposed auditing model are divided into two categories: quality management system criteria and quality assurance criteria. Each auditing criterion consists of several auditing sub-criteria and yardsticks that focus on the evidence that can be extracted to demonstrate process conformity to EU MDR requirements (more details can be found in Section 5.4: The Audit Criteria and Yardsticks).

The proposed auditing model could help medical device software development companies in their effort to achieve EU MDR certification (CE marking) and can help the software auditors of EU MDR to obtain evidence that demonstrates conformity to the EU MDR requirements, thus helping the software developers to follow the EU MDR requirements.

An evaluation was conducted to investigate the applicability of the proposed auditing model. The evaluation was performed based on the planning, examination, decision-making (PED) evaluation framework. Seven case studies were selected to be audited by the proposed auditing model to investigate whether or not they comply with the EU MDR requirements. As a result of the evaluation, no case study demonstrated full support of the auditing yardsticks of the proposed auditing model. However, the evidence gathered showed at least partial support for the requirements of EU MDR in each case study (more details can be found in Chapter 6: Evaluation of the Proposed Auditing Model).

7.2 Future work

The research presented in this thesis can lead to further work to improve our understanding of the EU MDR certification process for medical device software and our experience of the process in the context

of agile software organisations. In this thesis, several engineering models and frameworks have been investigated to enhance the early planning phase of Agile XP to accommodate important information for the EU MDR auditors. For example, the extended Agile XP user story and the proposed auditing model for EU MDR have been developed to help EU MDR auditors find auditing evidence in the context of agile software processes.

Accordingly, the following ideas could be recommendations for future work:

Agile practices within different medical devices software regulations

It could be interesting to consider the investigation and the enhancement of the agile practices within other healthcare regulations for medical device software such as the Medicines and Healthcare Products Regulatory Agency (MHRA) in the UK, the National Medical Product Administration (NMPA) in China, or the Food and Drug Administration (FDA) in the United States.

The extended Agile XP user story

The extended Agile XP user story could be improved by making the capturing requirements and other aspects fully automated. This could be done by integrating the Natural Language Processing (NLP) or other techniques of machine learning to convert the customer requirements that are given in a natural language into formal requirements. In addition, using the NLP would allow the classification of user story aspects such as functional requirements and non-functional requirements.

The auditing model

The proposed auditing model could be implemented in a way that would allow the auditors to automatically extract the required evidence from the documents and then classify the evidence based on the type, such as textual evidence, modelling evidence, graphical evidence, and coding evidence. To do so, there should be an integration of the ASM ground model and machine learning aspects such as the classification and detection of the NLP.

Bibliography

[1] McHugh, M., McCaffery, F., Fitzgerald, B., Stol, K.-J., Casey, V., Coady, G.: Balancing Agility and Discipline in a Medical Device Software Organisation. In: Woronowicz, T., Rout, T., O'Connor, R.V., and Dorling, A. (eds.) Software Process Improvement and Capability Determination. pp. 199–210. Springer Berlin Heidelberg, Berlin, Heidelberg (2013).

[2] Dybå, T., Dingsøyr, T.: Empirical studies of agile software development: A systematic review. Information and Software Technology. 50, 833–859 (2008). https://doi.org/10.1016/j.infsof.2008.01.006.

[3] McHugh, M., McCaffery, F., Casey, V.: Barriers to Adopting Agile Practices When Developing Medical Device Software. In: Mas, A., Mesquida, A., Rout, T., O'Connor, R.V., and Dorling, A. (eds.) Software Process Improvement and Capability Determination. pp. 141–147. Springer Berlin Heidelberg, Berlin, Heidelberg (2012).

[4] McHugh, M., McCaffery, F., Casey, V., Pikkarainen, M.: Integrating Agile Practices with a Medical Device Software Development Lifecycle. In: European Systems and Software Process Improvement and Innovation Conference, EuroSPI. , Vienna, Austria (2012).

[5] Demissie, S., Keenan, F., McCaffery, F.: Investigating the Suitability of Using Agile for Medical Embedded Software Development. In: Clarke, P.M., O'Connor, R.V., Rout, T., and Dorling, A. (eds.) Software Process Improvement and Capability Determination. pp. 409–416. Springer International Publishing, Cham (2016).

[6] Kaisti, M., Rantala, V., Mujunen, T., Hyrynsalmi, S., Könnölä, K., Mäkilä, T., Lehtonen, T.: Agile methods for embedded systems development-A literature review and a mapping study. EURASIP Journal on Embedded Systems. 2013, (2013). https://doi.org/10.1186/1687-3963-2013-15.

[7] Albuquerque, C.O., Antonino, P.O., Nakagawa, E.Y.: An Investigation into Agile Methods in Embedded Systems Development. In: Murgante, B., Gervasi, O., Misra, S., Nedjah, N., Rocha, A.M.A.C., Taniar, D., and Apduhan, B.O. (eds.) Computational Science and Its Applications – ICCSA 2012. pp. 576–591. Springer Berlin Heidelberg, Berlin, Heidelberg (2012).

[8] Cordeiro, L., Barreto, R., Barcelos, R., Jr, M., Lucena Jr, V., Maciel, P.: TXM: an agile HW/SW development methodology for building medical devices. ACM SIGSOFT Software Engineering Notes. 32, (2007).

[9] McCaffery, F., Lepmets, M., Clarke, P.: Medical Device Software as a Subsystem of an Overall Medical Device. Presented at the The First International Conference on Fundamentals and Advances in Software Systems Integration (FASSI 2015) , Venice, Italy (2015).

[10] Kaisti, M., Mujunen, T., Mäkilä, T., Rantala, V., Lehtonen, T.: Agile Principles in the Embedded System Development. In: Cantone, G. and Marchesi, M. (eds.) Agile Processes in Software Engineering and Extreme Programming. pp. 16–31. Springer International Publishing, Cham (2014).

[11] Özcan-Top, Ö., McCaffery, F.: To what extent the medical device software regulations can be achieved with agile software development methods? XP—DSDM—Scrum. The Journal of Supercomputing. 75, 5227–5260 (2019). https://doi.org/10.1007/s11227-019-02793-x.

[12] Spence, J.W.: There Has to Be a Better Way! Presented at the AGILE Conference , Denver, CO, USA (2005).

[13] McHugh, M., McCaffery, F., Coady, G.: An Agile Implementation within a Medical Device Software Organisation. In: Mitasiunas, A., Rout, T., O'Connor, R.V., and Dorling, A. (eds.) Software Process Improvement and Capability Determination. pp. 190–201. Springer International Publishing, Cham (2014).

[14] Cawley, O., Wang, X., Richardson, I.: Lean/Agile Software Development Methodologies in Regulated Environments - State of the Art. Lecture Notes in Business Information Processing. 65, (2010). https://doi.org/10.1007/978-3-642-16416-3_4.

[15] Digital.ai: The 14th Annual State of Agile Report, www.stateofagile.com, (2020).

[16] Acharya, B., Kumar Sahu, P.: SOFTWARE DEVELOPMENT LIFE CYCLE MODELS: A REVIEW PAPER. International Journal of Advanced Research in Engineering and Technology (IJARET). Volume 11, 169–176 (2020).

[17] Akinsola, J.E.T., Ogunbanwo, A.S., Okesola, O.J., Odun-Ayo, I.J., Ayegbusi, F.D., Adebiyi, A.A.: Comparative Analysis of Software Development Life Cycle Models (SDLC). In: Silhavy, R. (ed.) Intelligent Algorithms in Software Engineering. pp. 310–322. Springer International Publishing, Cham (2020).

[18] Shylesh S: A Study of Software Development Life Cycle Process Models. SSRN. (2017).

[19] Balaji, S., Murugaiyan, M.: WATEERFALLVs V-MODEL Vs AGILE: A COMPARATIVE STUDY ON SDLC. International Journal of Information Technology and Business Management. Vol.2, (2012).

[20] Govardhan, D.: A Comparison Between Five Models Of Software Engineering. IJCSI International Journal of Computer Science Issues 1694-0814. 7, 94–101 (2010).

[21] Alshamrani, A., Bahattab, A.: A Comparison Between Three SDLC Models Waterfall Model, Spiral Model, and Incremental/Iterative Model. IJCSI. Volume 12, (2015).

[22] Stoica, M., Mircea, M., Ghilic-Micu, B.: Software Development: Agile vs. Traditional. Informatica Economica. 17, 64–76 (2013). https://doi.org/10.12948/issn14531305/17.4.2013.06.

[23] Boehm, B.W.: A spiral model of software development and enhancement. Computer. 21, 61–72 (1988). https://doi.org/10.1109/2.59.

[24] Seema, S., Kute, S., Surabhi, D., Thorat: A Review on Various Software Development Life Cycle (SDLC) Models. 3, 2320–5156 (2014).

[25] Martin, R.C.: Agile Software Development: Principles, Patterns, and Practices. Prentice Hall PTR, USA (2003).

[26] Pressman, R.S.: Software engineering: a practitioner's approach. McGraw-Hill Education, New York, NY (2015).

[27] Ibrahim, N.: An Overview of Agile Software Development Methodology and Its Relevance to Software Engineering. Jurnal Sistem Informasi. 2, 69–80 (2007).

[28] Agile Manifesto, https://agilemanifesto.org/.

[29] Beck, K.: Extreme programming explained: embrace change. Addison-Wesley, Boston, MA (1999).

[30] VersionOne: 13th Annual State Of Agile Report, www.StateOfAgile.com, (2019).

[31] Khramtchenko, S.: Comparing extreme programming and feature driven development in academic and regulated environments. Presented at the CSCIE-275: Software Architecture and Engineering , Harvard University (2004).

[32] Qureshi, M.R.J.: Agile software development methodology for medium and large projects. IET Softw. 6, 358 (2012). https://doi.org/10.1049/iet-sen.2011.0110.

[33] L. Rising, N. S. Janoff: The Scrum software development process for small teams. IEEE Software. 17, 26–32 (2000). https://doi.org/10.1109/52.854065.

[34] Vlaanderen, K., Jansen, S., Brinkkemper, S., Jaspers, E.: The agile requirements refinery: Applying SCRUM principles to software product management. Information and Software Technology. 53, 58–70 (2011). https://doi.org/10.1016/j.infsof.2010.08.004.

[35] Palmer, S.R., Felsing, M.: A Practical Guide to Feature-Driven Development. Pearson Education (2001).

[36] Arbain, Adila Firdaus, Ghani, Imran, Jeong, Seung-Ryul: A Systematic Literature Review on Secure Software Development using Feature Driven Development (FDD) Agile Model. Journal of Internet Computing and Services. 15, 13–27 (2014). https://doi.org/10.7472/JKSII.2014.15.1.13.

[37] Boehm, B., Turner, R.: Management Challenges to Implementing Agile Processes in Traditional Development Organizations. Software, IEEE. 22, 30–39 (2005). https://doi.org/10.1109/MS.2005.129.

[38] Fitzgerald, B., Stol, K.-J., O'Sullivan, R., O'Brien, D.: Scaling agile methods to regulated environments: An industry case study. (2013).

[39] Stateofagile, https://stateofagile.com/.

[40] Rottier, P., Rodrigues, V.: Agile Development in a Medical Device Company. (2008). https://doi.org/10.1109/Agile.2008.52.

[41] R. Rasmussen, T. Hughes, J. R. Jenks, J. Skach: Adopting Agile in an FDA Regulated Environment. In: 2009 Agile Conference. pp. 151–155 (2009). https://doi.org/10.1109/AGILE.2009.50.

[42] K. Weyrauch: What are we arguing about? A framework for defining agile in our organization. In: AGILE 2006 (AGILE'06). p. 8 pp. – 220 (2006). https://doi.org/10.1109/AGILE.2006.62.

[43] C. K. Riemenschneider, B. C. Hardgrave, F. D. Davis: Explaining software developer acceptance of methodologies: a comparison of five theoretical models. IEEE Transactions on Software Engineering. 28, 1135–1145 (2002). https://doi.org/10.1109/TSE.2002.1158287.

[44] Alsaadi, M., Qasaimeh, M., Tedmori, S., Almakadmeh, K.: HIPAA Security and Privacy Rules Auditing in Extreme Programming Environments. International Journal of Information Systems in the Service Sector (IJISSS). 9, 1–21 (2017).

[45] Grenning, J.: Launching eXtreme programming at a process intensive company. Software, IEEE. 18, 27–33 (2001). https://doi.org/10.1109/52.965799.

[46] Cleland-Huang, J.: Traceability in Agile Projects. In: Cleland-Huang, J., Gotel, O., and Zisman, A. (eds.) Software and Systems Traceability. pp. 265–275. Springer London, London (2012). https://doi.org/10.1007/978-1-4471-2239-5_12.

[47] Kant, P., Hammond, K., Coutts, D., Chapman, J., Clarke, N., Corduan, J., Davies, N., Díaz, J., Güdemann, M., Jeltsch, W., Szamotulski, M., Vinogradova, P.: Flexible Formality Practical Experience with Agile Formal Methods. In: Byrski, A. and Hughes, J. (eds.) Trends in Functional Programming. pp. 94–120. Springer International Publishing, Cham (2020).

[48] Schwaber, K.: SCRUM Development Process. In: Sutherland, J., Casanave, C., Miller, J., Patel, P., and Hollowell, G. (eds.) Business Object Design and Implementation. pp. 117–134. Springer London, London (1997).

[49] Cho, J.: Issues and Challenges in Scrum Implementation. International Journal of Scientific and Engineering Research. VOL IX, (2008).

[50] G. Derbier: Agile development in the old economy. In: Proceedings of the Agile Development Conference, 2003. ADC 2003. pp. 125–131 (2003). https://doi.org/10.1109/ADC.2003.1231462.

[51] Alsaadi, M., Lisitsa, A., Qasaimeh, M.: Minimizing the ambiguities in medical devices regulations based on software requirement engineering techniques. In: Proceedings of the Second International Conference on Data Science, E-Learning and Information Systems, DATA 2019, Dubai, UAE, December 2-5, 2019. p. 18:1-18:5 (2019). https://doi.org/10.1145/3368691.3368709.

[52] Jeary, T., Schulze, K., Restuccia, D.: What medical writers need to know about regulatory approval of mobile health and digital healthcare devices. Medical Writing. 28, 28–33 (2019).

[53] European Commission: REGULATION (EU) 2017/745 OF THE EUROPEAN PARLIAMENT AND OF THE COUNCIL of 5 April 2017 on medical devices, amending Directive 2001/83/EC, Regulation (EC) No 178/2002 and Regulation (EC) No 1223/2009 and repealing Council Directives 90/385/EEC and 93/42/EEC. Official Journal of the European Union. 60, (2017).

[54] U.S. FOOD & DRUG ADMINISTRATION, https://www.fda.gov/medical-devices/device-advice-comprehensive-regulatory-assistance/overview-device-regulation.

[55] Wiersinga, J.: Regulation of Medical Digital Technologies. In: Marston, H.R., Freeman, S., and Musselwhite, C. (eds.) Mobile e-Health. pp. 277–295. Springer International Publishing, Cham (2017). https://doi.org/10.1007/978-3-319-60672-9_13.

[56] Atchinson, B., Fox, D.: The politics of the Health Insurance Portability and Accountability Act. Health Affairs. 16, pp.146-150 (1997).

[57] Medical Device Coordination Group: Guidance on Qualification and Classification of Software in Regulation (EU) 2017/745 – MDR and Regulation (EU) 2017/746 – IVDR. (2019).

[58] French-Mowat, E., Burnett, J.: How are medical devices regulated in the European Union? J R Soc Med. 105, 22–28 (2012). https://doi.org/10.1258/jrsm.2012.120036.

[59] McHugh, M., McCaffery, F., Casey, V.: Standalone Software as an Active Medical Device. In: O'Connor, R.V., Rout, T., McCaffery, F., and Dorling, A. (eds.) Software Process Improvement and Capability Determination. pp. 97–107. Springer Berlin Heidelberg, Berlin, Heidelberg (2011).

[60] M. Zema, S. Rosati, V. Gioia, M. Knaflitz, G. Balestra: Developing medical device software in compliance with regulations. In: 2015 37th Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC). pp. 1331–1334 (2015). https://doi.org/10.1109/EMBC.2015.7318614.

[61] T. D. Breaux, M. W. Vail, A. I. Anton: Towards Regulatory Compliance: Extracting Rights and Obligations to Align Requirements with Regulations. In: 14th IEEE International Requirements Engineering Conference (RE'06). pp. 49–58 (2006). https://doi.org/10.1109/RE.2006.68.

[62] Reidenberg, J.R., Bhatia, J., Breaux, T.D., Norton, T.B.: Ambiguity in Privacy Policies and the Impact of Regulation. The Journal of Legal Studies. 45, S163–S190 (2016).

[63] A. K. Massey, R. L. Rutledge, A. I. Antón, P. P. Swire: Identifying and classifying ambiguity for regulatory requirements. In: 2014 IEEE 22nd International Requirements Engineering Conference (RE). pp. 83–92 (2014). https://doi.org/10.1109/RE.2014.6912250.

[64] Maxwell, J.C.: Reasoning About Legal Text Evolution for Regulatory Compliance in Software Systems, https://liverpool.idm.oclc.org/login?url?url=https://www.proquest.com/dissertations-theses/reasoning-about-legal-text-evolution-regulatory/docview/1462054954/se-2?accountid=12117, (2013).

[65] P. N. Otto, A. I. Anton: Addressing Legal Requirements in Requirements Engineering. In: 15th IEEE International Requirements Engineering Conference (RE 2007). pp. 5–14 (2007). https://doi.org/10.1109/RE.2007.65.

[66] S. Ghaisas, A. Sainani, P. R. Anish: Resolving Ambiguities in Regulations: Towards Achieving the Kohlbergian Stage of Principled Morality. In: 2018 IEEE/ACM 40th International Conference on Software Engineering: Software Engineering in Society (ICSE-SEIS). pp. 57–60 (2018).

[67] Jackson, J.: The Costs of Medical Privacy Breach. MD advisor: a journal for New Jersey medical community. 8, 4–12 (2015).

[68] Davis, A.M.: Software requirements: objects, functions, and states. Prentice-Hall, Inc. (1993).

[69] Schneider, G.M., Martin, J., Tsai, W.T.: An Experimental Study of Fault Detection in User Requirements Documents. ACM Trans. Softw. Eng. Methodol. 1, 188–204 (1992). https://doi.org/10.1145/128894.128897.

[70] Berry, D.M., Kamsties, E., Krieger, M.M.: From Contract Drafting to Software Specification: Linguistic Sources of Ambiguity- A Handbook. Computer Science Department University of Waterloo, Canada (2003).

[71] Sommerville, I.: Software engineering. Pearson, Boston (2011).

[72] Bushnell, D.S.: Input, process, output: a model for evaluating training, (1990).

[73] Siau, K., Lee, L.: Are use case and class diagrams complementary in requirements analysis? An experimental study on use case and class diagrams in UML. Requirements Engineering. 9, 229–237 (2004). https://doi.org/10.1007/s00766-004-0203-7.

[74] R. Razali, P. Najafi, S. H. Mirisaee: Combining Use Case Diagram and Integrated Definition's IDEFO — A preliminary study. In: The 2nd International Conference on Software Engineering and Data Mining. pp. 231–236 (2010).

[75] Börger, E., Gargantini, A., Riccobene, E.: Abstract State Machines 2003: Advances in Theory and Practice 10th International Workshop, ASM 2003 Taormina, Italy, March 3 7, 2003 Proceedings. Springer-Verlag Berlin Heidelberg, Berlin; Heidelberg (2003).

[76] Lucassen, G., Dalpiaz, F., Werf, J.M.E.M. van der, Brinkkemper, S.: The Use and Effectiveness of User Stories in Practice. In: Daneva, M. and Pastor, O. (eds.) Requirements Engineering: Foundation for Software Quality. pp. 205–222. Springer International Publishing, Cham (2016).

[77] Cohn, M.: User stories applied: for agile software development. Addison-Wesley, Boston (2004).

[78] Carniel, C.A., Pegoraro, R.A.: Metamodel for Requirements Traceability and Impact Analysis on Agile Methods. In: Santos, V.A. dos, Pinto, G.H.L., and Serra Seca Neto, A.G. (eds.) Agile Methods. pp. 105–117. Springer International Publishing (2018).

[79] Lucassen, G., Dalpiaz, F., van der Werf, J.M.E.M., Brinkkemper, S.: Improving agile requirements: the Quality User Story framework and tool. Requirements Engineering. 21, 383–403 (2016). https://doi.org/10.1007/s00766-016-0250-x.

[80] Mazni, O., Sharifah-Lailee, S.-A., Azman, Y.: Agile Documents: Toward Successful Creation of Effective Documentation. In: Sillitti, A., Martin, A., Wang, X., and Whitworth, E. (eds.) Agile Processes in Software Engineering and Extreme Programming. pp. 196–201. Springer Berlin Heidelberg (2010).

[81] Hoda, R., Noble, J., Marshall, S.: Documentation Strategies on Agile Software Development Projects. International Journal of Agile and Extreme Software Development (IJAESD). 1, (2012). https://doi.org/10.1504/IJAESD.2012.048308.

[82] Börger, E., Raschke, A.: Modeling companion for software practitioners. Springer-Verlag Berlin Heidelberg (2018).

[83] Börger, E.: Why Programming Must Be Supported by Modeling and How. In: Margaria, T. and Steffen, B. (eds.) Leveraging Applications of Formal Methods, Verification and Validation. Modeling. pp. 89–110. Springer International Publishing (2018).

[84] Rout, T.P.: ISO/IEC 15504 and Spice. In: Marciniak, J.J. (ed.) Encyclopedia of Software Engineering. p. sof171. John Wiley & Sons, Inc., Hoboken, NJ, USA (2002). https://doi.org/10.1002/0471028959.sof171.

[85] BS ISO/IEC 25030:2019 Systems and software engineering. Systems and software quality requirements and evaluation (SQuaRE). Quality requirements framework. BSI Standards Limited (2019).

[86] BS ISO/IEC 25010:2011 Systems and software engineering. Systems and software quality requirements and evaluation (SQuaRE). System and software quality models. BSI Standards Limited (2011).

[87] Bourque, P., Fairley, R.E., IEEE Computer Society: Guide to the software engineering body of knowledge. (2014).

[88] Glinz, M., Wieringa, R.: Guest Editors' Introduction: Stakeholders in Requirements Engineering. IEEE Softw. 24, 18–20 (2007). https://doi.org/10.1109/MS.2007.42.

[89] Alashqar, A., Elfetouh, A., El-Bakry, H.: Requirement Engineering for Non-Functional Requirements. International Journal of Information and Communication Technology Research. 5, 21–27 (2015).

[90] Umar, M., Naeem Ahmed Khan: Analyzing Non-Functional Requirements (NFRs) for software development. In: 2011 IEEE 2nd International Conference on Software Engineering and Service Science. pp. 675–678. IEEE, Beijing, China (2011). https://doi.org/10.1109/ICSESS.2011.5982328.

[91] Ngo-The, A., Ruhe, G.: Decision Support in Requirements Engineering. In: Aurum, A. and Wohlin, C. (eds.) Engineering and Managing Software Requirements. pp. 267–286. Springer Berlin Heidelberg, Berlin, Heidelberg (2005). https://doi.org/10.1007/3-540-28244-0_12.

[92] Firesmith, D.: Prioritizing Requirements. JOT. 3, 35 (2004). https://doi.org/10.5381/jot.2004.3.8.c4.

[93] Berander, P., Andrews, A.: Requirements Prioritization. In: Aurum, A. and Wohlin, C. (eds.) Engineering and Managing Software Requirements. pp. 69–94. Springer Berlin Heidelberg, Berlin, Heidelberg (2005). https://doi.org/10.1007/3-540-28244-0_4.

[94] Hudaib, A., Masadeh, R., Haj Qasem, M., Alzaqebah, A.: Requirements Prioritization Techniques Comparison. Modern Applied Science. 12, (2018). https://doi.org/10.5539/mas.v12n2p62.

[95] Dahlstedt, Å.G., Persson, A.: Requirements Interdependencies: State of the Art and Future Challenges. In: Aurum, A. and Wohlin, C. (eds.) Engineering and Managing Software Requirements. pp. 95–116. Springer Berlin Heidelberg, Berlin, Heidelberg (2005). https://doi.org/10.1007/3-540-28244-0_5.

[96] P. Carlshamre, K. Sandahl, M. Lindvall, B. Regnell, J. Natt och Dag: An industrial survey of requirements interdependencies in software product release planning. In: Proceedings Fifth IEEE International Symposium on Requirements Engineering. pp. 84–91 (2001). https://doi.org/10.1109/ISRE.2001.948547.

[97] Khalaf, M.: MACHINE LEARNING APPROACHES AND WEB-BASED SYSTEM TO THE APPLICATION OF DISEASE MODIFYING THERAPY FOR SICKLE CELL. Liverpool John Moores University, Liverpool, UK (2018).

[98] BS EN ISO 9000:2015 Quality management systems. Fundamentals and vocabulary. BSI.

[99] Scriven, M.: Evaluation thesaurus. Sage Publications, Newbury Park, Calif (1991).

[100] Guidance for notified bodies on the use of MDSAP audit reports in the context of surveillance audits carried out under the Medical Devices Regulation (MDR)/In Vitro Diagnostic medical devices Regulation (IVDR). (2020).

[101] Lopez, M.: An Evaluation Theory Perspective of the Architecture Tradeoff Analysis Method (ATAM). Carnegie MelIon Software Engineering Institute (2000).

[102] Qasaimeh, M., Abran, A.: An Audit Model for ISO 9001 Traceability Requirements in Agile-XP Environments. JSW. 8, 1556–1567 (2013). https://doi.org/10.4304/jsw.8.7.1556-1567.

[103] ZAROUR, M.: METHODS TO EVALUATE LIGHTWEIGHT SOFTWARE PROCESS ASSESSMENT METHODS BASED ON EVALUATION THEORY AND ENGINEERING DESIGN PRINCIPLES, https://espace.etsmtl.ca/id/eprint/92/, (2009).

[104] House, E. r.: Evaluating With Validity. Sage Publications, London (1980).

[105] BS ISO/IEC/IEEE 12207-2:2020 Systems and software engineering. Software life cycle processes. (2020).

[106] BS EN ISO 13485:2016 - TC Tracked Changes. Medical devices. Quality management systems. Requirements for regulatory purposes. BSI (2016).

[107] BS EN ISO 14971:2019 - TC Tracked Changes. Medical devices. Application of risk management to medical devices. BSI (2019).

[108] Stracke, C.: Process-oriented quality management. In: Ehlers, U.-D. and Pawlowski, J.M. (eds.) Handbook on Quality and Standardisation in E-Learning. pp. 79–96. Springer Berlin Heidelberg, Berlin, Heidelberg (2006). https://doi.org/10.1007/3-540-32788-6_6.

[109] McCall, J., Richards, P.A., Walters, G.: Factors in software quality: concept and definitions of software quality. Presented at the (1977).

[110] Duncan, R.: The Quality of Requirements in Extreme Programming. Presented at the , Mississippi State University (2001).

[111] Pressman, R.S., Maxim, B.R.: Software engineering: a practitioner's approach. (2020).

[112] S. I. Hashmi, J. Baik: Software Quality Assurance in XP and Spiral - A Comparative Study. In: 2007 International Conference on Computational Science and its Applications (ICCSA 2007). pp. 367–374 (2007). https://doi.org/10.1109/ICCSA.2007.65.

[113] A. Albadarneh, I. Albadarneh, A. Qusef: Risk management in Agile software development: A comparative study. In: 2015 IEEE Jordan Conference on Applied Electrical Engineering and Computing Technologies (AEECT). pp. 1–6 (2015). https://doi.org/10.1109/AEECT.2015.7360573.

[114] Jeffries, R., Anderson, A., Hendrickson, C.: Extreme programming installed. Addison-Wesley, Boston (2001).

[115] D. R. Wallace, R. U. Fujii: Software verification and validation: an overview. IEEE Software. 6, 10–17 (1989). https://doi.org/10.1109/52.28119.

[116] Ming Huo, Verner, J., Liming Zhu, Babar, M.A.: Software quality and agile methods. In: Proceedings of the 28th Annual International Computer Software and Applications Conference, 2004. COMPSAC 2004. pp. 520–525 vol.1 (2004). https://doi.org/10.1109/CMPSAC.2004.1342889.

[117] Arcaini, P., Gargantini, A., Riccobene, E.: AsmetaSMV: A Way to Link High-Level ASM Models to Low-Level NuSMV Specifications. In: Frappier, M., Glässer, U., Khurshid, S., Laleau, R., and Reeves, S. (eds.) Abstract State Machines, Alloy, B and Z. pp. 61–74. Springer Berlin Heidelberg, Berlin, Heidelberg (2010).

[118] Arcaini, P., Bonfanti, S., Gargantini, A., Riccobene, E., Scandurra, P.: Modelling an Automotive Software-Intensive System with Adaptive Features Using ASMETA. In: Raschke, A., Méry, D., and Houdek, F. (eds.) Rigorous State-Based Methods. pp. 302–317. Springer International Publishing, Cham (2020).

[119] AsmetaL syntax A Quick Guide, https://asmeta.github.io/material/AsmetaL_quickguide.html.

[120] Lessambo, F.I.: Audit Evidence and Documentation. In: Lessambo, F.I. (ed.) Auditing, Assurance Services, and Forensics: A Comprehensive Approach. pp. 155–182. Springer International Publishing, Cham (2018). https://doi.org/10.1007/978-3-319-90521-1_9.

[121] Phung, K., Odongo, S., Ogunshile, E.: Covid-19 Care – A mobile application to help connect volunteers and vulnerable people in the community during the Covid-19 lockdown. In: 2020 8th International Conference in Software Engineering Research and Innovation (CONISOFT). pp. 124–133 (2020). https://doi.org/10.1109/CONISOFT50191.2020.00027.

[122] Zhang, H., Dimitrov, D., Simpson, L., Plaks, N., Singh, B., Penney, S., Charles, J., Sheehan, R., Flammini, S., Murphy, S., Landman, A.: A Web-Based, Mobile-Responsive Application to Screen Health Care Workers for COVID-19 Symptoms: Rapid Design, Deployment, and Usage. JMIR Form Res. 4, e19533 (2020). https://doi.org/10.2196/19533.

[123] Özcan, T., Kocak, S., Brune, P.: Agile Software Development with Open Source Software in a Hospital Environment – Case Study of an eCRF-System for Orthopaedical Studies. In: Daniel, F., Dolog, P., and Li, Q. (eds.) Web Engineering. pp. 439–451. Springer Berlin Heidelberg, Berlin, Heidelberg (2013).

[124] Gary, K., Enquobahrie, A., Ibanez, L., Cheng, P., Yaniv, Z., Cleary, K., Kokoori, S., Muffih, B., Heidenreich, J.: Agile methods for open source safety-critical software. Software: Practice and Experience. 41, 945–962 (2011). https://doi.org/10.1002/spe.1075.

[125] Narasimhadevara, A., Radhakrishnan, T., Leung, B., Jayakumar, R.: On designing a usable interactive system to support transplant nursing. Journal of Biomedical Informatics. 41, 137–151 (2008). https://doi.org/10.1016/j.jbi.2007.03.006.

[126] I Gusti Ngurah Suryantara, Johanes Andry: Development of Medical Record With Extreme Programming SDLC. IJNMT. 5, (2018). https://doi.org/10.31937/ijnmt.v5i1.706.

Appendices

Appendix A

The description of requirements below are as stated in the Official Journal of the European Union [53]:

ANNEX I: GENERAL SAFETY AND PERFORMANCE REQUIREMENTS

ANNEX I, CHAPTER I: GENERAL REQUIREMENTS:

- 3. *Manufacturers shall establish, implement, document and maintain a risk management system. Risk management shall be understood as a continuous iterative process throughout the entire lifecycle of a device, requiring regular systematic updating. In carrying out risk management manufacturers shall:*

    1. *establish and document a risk management plan for each device;*
    2. *identify and analyse the known and foreseeable hazards associated with each device;*

ANNEX I, CHAPTER II: REQUIREMENTS REGARDING DESIGN AND MANUFACTURE:

- 17.1. *Devices that incorporate electronic programmable systems, including software, or software that are devices in themselves, shall be designed to ensure repeatability, reliability and performance in line with their intended use. In the event of a single fault condition, appropriate means shall be adopted to eliminate or reduce as far as possible consequent risks or impairment of performance.*
- 17.2. *For devices that incorporate software or for software that are devices in themselves, the software shall be developed and manufactured in accordance with the state of the art, taking into account the principles of development life cycle, risk management, including information security, verification and validation.*
- 17.4. *Manufacturers shall set out minimum requirements concerning hardware, IT network characteristics and IT security measures, including protection against unauthorised access, necessary to run the software as intended.*

ANNEX I, CHAPTER III: REQUIREMENTS REGARDING THE INFORMATION SUPPLIED WITH THE DEVICE:

- 23.1. *General requirements regarding the information supplied by the manufacturer. Each device shall be accompanied by the information needed to identify the device and its manufacturer, and by any safety and performance information relevant to the user, or any other person, as appropriate.*

ANNEX II: TECHNICAL DOCUMENTATION

ANNEX II: 1. DEVICE DESCRIPTION AND SPECIFICATION, INCLUDING VARIANTS AND ACCESSORIES

- 1.1. *Device description and specification;*
- 1.1.a. *product or trade name and a general description of the device including its intended purpose and intended users;*
- 1.1.b. *Principles of operation of the device and its mode of action, scientifically demonstrated if necessary;*
- 1.1.g. *An explanation of any novel features;*
- 1.1.h. *a description of the accessories for a device, other devices and other products that are not devices, which are intended to be used in combination with it;*
- 1.1.j. *A general description of the key functional elements, e.g. its parts/components (including software if appropriate), its formulation, its composition, its functionality and, where relevant, its qualitative and quantitative composition. Where appropriate, this shall include labelled pictorial representations (e.g. diagrams, photographs, and drawings), clearly indicating key parts/components, including sufficient explanation to understand the drawings and diagrams;*
- 1.1.1. *Technical specifications, such as features, dimensions and performance attributes, of the device and any variants/configurations and accessories that would typically appear in the product specification made available to the user, for example in brochures, catalogues and similar publications.*

ANNEX II: 3. DESIGN AND MANUFACTURING INFORMATION

- 3.b. *complete information and specifications, including the manufacturing processes and their validation, their adjuvants, the continuous monitoring and the final product testing. Data shall be fully included in the technical documentation.*

ANNEX II: 4. GENERAL SAFETY AND PERFORMANCE REQUIREMENTS

The documentation shall contain information for the demonstration of conformity with the general safety and performance requirements set out in Annex I.: Annex I. 17.1, Annex I. 17.2, and Annex I. 17.4.

ANNEX II: 5. BENEFIT-RISK ANALYSIS AND RISK MANAGEMENT

- The documentation shall contain information on:
    1. *The benefit-risk analysis referred to in Sections 1 and 8 of Annex I, and*
    2. *The solutions adopted and the results of the risk management referred to in Section 3 of Annex I.*

ANNEX II: 6. PRODUCT VERIFICATION AND VALIDATION

The documentation shall contain the results and critical analyses of all verifications and validation tests and/or studies undertaken to demonstrate conformity of the device with the requirements of this Regulation and in particular the applicable general safety and performance requirements

- 6.1.(b) *detailed information regarding test design, complete test or study protocols, methods of data analysis, in addition to data summaries and test conclusions regarding in particular:*

    1. *Software verification and validation (describing the software design and development process and evidence of the validation of the software, as used in the finished device).*

ANNEX IX: CONFORMITY ASSESSMENT BASED ON A QUALITY MANAGEMENT SYSTEM AND ON ASSESSMENT OF TECHNICAL DOCUMENTATION

ANNEX IX: 1. *The manufacturer shall establish, document and implement a quality management system as described in Article 10(9) and maintain its effectiveness throughout the life cycle of the devices concerned.*

ANNEX IX: 2. *Quality management system assessment:*

- *the documentation on the manufacturer's quality management system.*
- *a documented description of the procedures in place to fulfil the obligations arising from the quality management system and required under this Regulation and the undertaking by the manufacturer in question to apply those procedures.*
- *a description of the procedures in place to ensure that the quality management system remains adequate and effective, and the undertaking by the manufacturer to apply those procedures.*

Appendix B

The ASM executable code for the proposed auditing model:

```
asm AudtingModel
import ../../STDL/StandardLibrary

signature:
abstract domain Docs
abstract domain Yardsticks
abstract domain DataType
abstract domain ExtractedEvidence
enum domain State = { AWAITDOCUMENT |
EXTRACTINGEVIDENCE | ASSIGNINGEVIDENCE | MAKINGDECISION }
enum domain EvidenceTypes = {TEXTUAL | MODELLING | GRAPHICAL
| CODING}
enum domain YardStickClass = {Y1|Y2 | Y3 | Y4 |Y5|Y6 | Y7 | Y8|Y9
| Y10 | Y11}
dynamic controlled currDocument: Docs
dynamic controlled audtingModelState: State
dynamic controlled uploadDocuments : Docs
dynamic controlled extractEvidence: evidenceTypes
dynamic controlled classifyEvidenceType : EvidenceTypes
dynamic controlled assignEvidence: Yardsticks
dynamic controlled getMaxPredictedClass: Real
dynamic controlled giveDecision: Boolean
dynamic controlled showMessage: Any
derived supported: Prod (Yardsticks, Evidence) = Boolean
static image: DataType
static text: DataType
static model: DataType
static code: DataType
definitions:
macro rule r_uploadtDocuments =
if ( audtingModelState = AWAITDOCUMENT) then
if ( $c in Docs with $c= uploadDocuments) then
```

```
par
currDocument := uploadDocuments
audtingModelState := EXTRACTINGEVIDENCE
showMessage := "Please wait while finding the evidence"
endpar
endif
endif
macro rule r_ extractEvidence =

if ( audtingModelState = EXTRACTINGEVIDENCE) then

      if ( $g in DataType = image) then
par
classifyEvidenceType := GRAPHICAL, $g
endpar
endif
else if ($t in DataType = text) than
par
classifyEvidenceType := TEXTUAL, $t
endpar
        end else if
else if ($m in Datatype= model) than
par
classifyEvidenceType := MODELLING, $m
endpar
endelseif
else if ($c in DataType = code) than
par
classifyEvidenceType := CODES, $c
endpar
endelseif
par
audtingModelState := ASSIGNINGEVIDENCE
showMessage := "Please wait while auditing model assigning
evidence to yardsticks"
endpar
                      endif
macro rule r_ assignEvidence =

if ( audtingModelState = ASSIGNINGEVIDENCE) then

If ($e in ExtractedEvidence with YardStickClass.getMaxPredictedClass
=>0.8) then
par
               $e: = YardStickClass
```

```
audtingModelState := MAKINGDECISION
showMessage := "Please wait for the decision"
endpar
                    endif
 endif
main rule r_ giveDecision =
if ( audtingModelState = MAKINGDECISION) then
if ( ExtractedEvidence = YardStickClass) then
par
giveDecision := supported
showMessage := "Extracted evidence meets the auditing yardsticks"
endpar
else
par
r_uploadtDocuments [ ]
r_ extractEvidence [ ]
r_ assignEvidence [ ]
showMessage := "Extracted evidence do not meet the auditing yardsticks"
endpar
endif
endif
default init s0:
        function audtingModelState = AWAITDOCUMENT
```