Lessons Learned from the Development of a Mechanical Ventilator for COVID-19

Andrea Bombarda*, Silvia Bonfanti*, Cristiano Galbiati¶, Angelo Gargantini*, Patrizio Pelliccione†, Elvinia Riccobene‡, Masayuki Wada‡‡

**University of Bergamo*, Bergamo, Italy, {andrea.bombarda,silvia.bonfanti,angelo.gargantini}@unibg.it
†*Gran Sasso Science Institute (GSSI)*, L'Aquila, Italy |
Chalmers | *University of Gothenburg*, Gothenburg, Sweden, patrizio.pelliccione@gssi.it
‡*Università degli Studi di Milano*, Milano, Italy, elvinia.riccobene@unimi.it
¶*Princeton University*, Princeton, USA | *Gran Sasso Science Institute*, L'Aquila |
INFN Laboratori Nazionali del Gran Sasso, Assergi, Italy, galbiati@princeton.edu
‡‡ *AstroCeNT, N. Copernicus Astronomical Center, Polish Academy of Sciences*, Warsaw, Poland, masayuki@camk.edu.pl

Abstract—**During the COVID-19 pandemic, many researchers all over the world have offered their time and competencies to face the heavy consequences of the disease. This is the case of a group of physicists, engineers, and physicians that around the middle of March 2020 started to develop a simplified mechanical lung ventilator, called MVM (Mechanical Ventilator Milano), to answer the high request of ventilators for Acute Respiratory Distress Syndrome (ARDS) in intensive care units. A prototype was ready in around one month.**

Since medical software malfunctions can lead to injuries or death of patients, before marketing MVM ventilators and distributing them in hospitals, software certification in accordance with the IEC 62304 standard was mandatory to guarantee system reliability. The team was then complemented by computer scientists specifically devoted to this task. The software re-engineering process, which lasted around two months from the end of the prototype, brought to a strong re-implementation of the device software components, which involved all the stakeholders in a continuous integration setting.

In this paper, we report the experience of the MVM control SW re-engineering necessary to show evidence that the SW adheres to the standards and to consequently obtain the certification. We share results and lessons learned from this *social* project, where more than 100 volunteer researchers worked towards software certification at the extreme of their strength to get a real device finished in a rush since strongly required to support physicians in treating COVID-19 patients.

I. Introduction

The development of safety-critical devices considers certification as a mandatory step [1]–[3] since a software failure or malfunctioning can compromise the health of human beings that interact with it. Examples of safety-critical systems certification can be found in avionics [4]–[6], robotic applications [7], cyber-physical systems [8], and healthcare [9].

In healthcare, software certification is required when the software is itself a medical device [10]–[14] and it is regulated by the FDA Guidelines [15] and the `IEC 62304` [16]. Medical system certification is a complex process, which has been addressed using different solutions. Even if using formal methods for software certification is a viable solution [17]–[19], some issues about their use are still open and they are not widely used in practice, except for model-based techniques [20].

The importance of a well-documented and correct software life cycle in the development of medical systems has been discussed in [21] and [22]. It is not limited to just the safety and the certification of medical devices [23], since adopting frameworks and the good principles of software engineering can help developers to compare different medical devices for identifying gaps between them and improve the capabilities of products [24]. The focus on well-documented and regulated frameworks poses some constraints on the adoption of agile software development or requires adaptation of selected agile methods and practices [25]. In the literature, we can find attempts to apply agile practices also in medical software development planning, for instance by integrating the more classical V-model with the agile one [26], [27].

In this paper, we aim at investigating whether the bunch of knowledge and experience we, as a community, acquired in the last years is sufficient for producing software devices that are compliant with safety standards when we are under emergency. By emergency we mean producing software under these two constraints: 1) the first hard constraint is time, meaning that the software device should be produced as soon as possible[1]; 2) the second hard constraint concerns establishing a development team in a hurry, in an emergent and voluntary fashion, based on the personal network, heterogeneous under various dimensions, and composed by people that dedicate their private time to the project, while still continuing their normal job. In other emergency situations, like hurricanes or earthquakes, there can be additional constraints like lack of energy power or Internet connection. However, in the context of this paper, we limit ourselves to the two constraints above, which are those that we observed in the experience we report.

Specifically, the research questions we want to answer are:

[1]Indeed, we might argue that time is a constraint for every company and for the production of almost every product. However, sometimes there are (emergency) situations that push even more this constraint.

- **RQ1:** Which development process is more appropriate for the development of safety-critical devices under emergency?
- **RQ2:** How the activities of the development process can be performed in order to be simplified and sped up under emergency?
- **RQ3:** How to deal with a heterogeneous development team built in emergent and voluntary fashion?

We provide an answer to these research questions by reporting our experience in developing an electro-mechanical ventilator for COVID-19, intended to provide ventilation support for patients that are in intensive therapy and that require mechanical ventilation. The produced mechanical lung ventilator, called MVM (Mechanical Ventilator Milano)[2], aims to be reliable, easily reproducible on a large scale, available in a short amount of time, and at a limited cost [28], around 1/5 with respect to other ventilators. The project started from an idea of the physicist Cristiano Galbiati, who was also the leader of the project, and saw, among others, the participation of Nobel prize Arthur McDonald, Canadian leader of the MVM Consortium. More precisely, experience results reported here concern re-engineering of the MVM software, necessary to adhere the software to the medical standards and obtain the certification. This required a serious rework on the overall software development, including requirements, architecture, design, testing, and so on. The effort and time required to certify and, consequently, re-engineer the application largely overcome the effort and time to build the initial version of the mechanical ventilator. We share results and lessons learned from this *social* project where more than 100 researchers having different knowledge and expertise, worked hard and fast in a strong collaborative manner to manage the software certification and to get a real device finished in a rush since strongly required to support physicians in treating COVID-19 patients. Note that almost all the team members have worked on a voluntary basis and monetary compensation has never been expected.

The paper is structured as follows. Sect. II presents the methodology we have used in order to provide an answer to our research questions and a brief case description. Sect. III presents the `IEC 62304` which has guided the development of the MVM ventilator, while in Sect. IV we present the software life-cycle that has been followed during this project and how it has been applied. Sect. V describes in detail all the development phases and the lessons learned during each of them, and Sect. VI presents related works in which software development and emergency situations are combined. Finally, Sect. VII discusses our lessons learned, by mapping each of them to the research questions, and concludes the paper.

II. Research Methodology

In order to provide an answer to our research questions, we conducted a *case study* [29], since it is a suitable method for studying a phenomenon in its natural context, especially when the phenomenon is difficult to study in isolation. Indeed, in our case, it is difficult to clearly and precisely identify and delimit in a real context the process and the activities to be followed when producing software devices that are supposed to be compliant to safety standards under emergency.

A. Case Description

MVM [30] is an electro-mechanical ventilator equivalent to the old and reliable Manley Ventilator [31]. It requires a source of compressed oxygen and medical air, that are readily available in intensive care units.

MVM is intended to provide ventilation support for patients that are in intensive therapy and that require mechanical ventilation. There are different modes of ventilation; the most common are volume-controlled and pressure-controlled. In the first mode, the respiratory time cycle is based on the required volume, while in the second mode the respiratory time cycle is controlled by the pressure. MVM works in pressure mode and, in particular, it implements two operative modes: Pressure Controlled Ventilation (PCV) and Pressure Support Ventilation (PSV). In the PCV mode, the respiratory cycle is kept constant and the pressure level changes between the target inspiratory pressure and the positive end-expiratory pressure. New inspiration is initiated either after a breathing cycle is over, or when the patient spontaneously initiates a breath. In the former case, the breathing cycle is controlled by two parameters: the respiratory rate and the ratio between the inspiratory and expiratory times. In the latter case, a spontaneous breath is triggered when the MVM detects a sudden pressure drop within the trigger window during expiration. The PSV mode is not suitable for patients that are not able to start breathing on their own because the respiratory cycle is controlled by the patient, while MVM partially takes over the work of breathing. A new respiratory cycle is initiated with the inspiratory phase, detected by the ventilator when a sudden drop in pressure occurs. When the patient's inspiratory flow drops below a set fraction of the peak flow, MVM stops the pressure support, thus allowing exhalation. If a new inspiratory phase is not detected within a certain amount of time (apnea lag), MVM will automatically switch to the PCV mode because it is assumed that the patient is not able to breathe alone.

A prototype of the MVM device was built in around one month and was available in the middle of April 2020. Till that moment, the team working on MVM only included physicists, physicians, and engineers as technicians and no well-defined SW development process was applied, no documentation was produced, and the certification was not even pursued. However, as usually required by the governments of countries where a medical device has to be marketed, the software certification is mandatory before selling and using the device. It became evident that the software has the actual responsibility for the movements of the mechanical parts and, therefore, there should be evidence for its correct functioning. Hence, a software task force, a group of computer scientists (including the authors), has been created to manage the certification of the software. We had a confirmation of what Marc Andreessen said in

[2]https://mvm.care/

Activity	# People	Deliverables
System development plan	3	5
Supporting activities	22	12
System requirements	5	1
Software Architecture Design & Risk Management	10	3
Software Requirement Specification & Software Detailed Design	21	15
Implementation	18	N/A
Unit Testing	22	20
Integration Testing	11	2
Validation Testing	9	2

TABLE I: Summary of the effort required for each phase

2011 [32]: "*Software Is Eating The World*". This has been already observed in other fields, like in automotive [33], and it is becoming evident the importance and the critical role of the software.[3]

The need for certification brought to a strong re-engineering of the MVM software components, while the hardware components required only minimal changes. The whole process lasted around two months and it involved, in a continuous integration manner, all the stakeholders. The first step of the process has concerned the analysis of all the standards that must be followed during the development of a medical device, and the identification of those related to MVM. Then, a plan and a process for all the activities to be performed have been devised. Starting from the software requirement specification, we have redefined the software architecture and we have rewritten some of the MVM components. Moreover, before the deployment of the software, we have performed all the testing activities (unit, integration, and validation testing) required by the standards.

The MVM software is released under MIT Licence[4], and it is stored in a private repository to control access to it and avoid incorrect use. Initially, all the documents and the code were publicly released, but, under the advice of a legal office, the team decided to limit access to them in order to avoid the risk of possible litigations.

B. Data Collection and Analysis

The role of the authors of this paper has been to contribute to the definition of the certification process; they also actively worked in many phases including requirements engineering, architectural design, testing (unit, integration, and validation), implementation, documentation, and traceability checking among the various artifacts and documents.

To provide an overview of the project activities that created the data for this case study, Table I reports (i) the activities, (ii) the documentation, in terms of the number of deliverables – each of these is in most cases a Microsoft Word document –, and (iii) the number of people involved in each activity.

Data themselves are diverse. They consist of the various deliverables created for the certification purpose, but also of work items created during the project completion, such as whiteboards sketches, notes (personal or shared within subgroups), and emails. Deliverables are all stored in a Google Drive shared folder to which the authors have full access. The authors of this paper collected independently those documents that are not stored into a folder shared among the project, merged, and cleaned, so to be used for the purpose of this paper. Moreover, the authors have also full access to the entire source code of the project, comments in the code, changes requests, test reports, and so on.

In addition to that, for two months within the project, we had a large number of virtual plenary and subgroups meetings, which have allowed us to get a look at the overall picture of the whole project and the single activities.

The practical experience we report in this paper is based on the data summarized above, which can be considered the starting point for identifying the development process, the various phases composing it, as well as relevant lessons learned that we will discuss in the paper. We aim to offer our experience for reducing risks in future developments in similar contexts. While collecting the lessons learned we made an effort to report and identify both positive and negative results.

C. Validity Analysis

The data described in the previous subsection were created not exactly for the purpose of proposing a general process and guidelines, i.e. for this paper. Instead, data were created for developing and producing a specific product, MVM, compliant with safety standards and under hard time constraints. This limitation cannot be removed by the fact that we had a diverse and rich amount of data to be analyzed. To handle this limitation, we use our direct experience and notes we kept throughout the project.

All the authors extensively discussed the interpretation of the data to achieve observer triangulation. In line with our decision for conducting a case study, i.e. to study a phenomenon in its natural context, our findings are indeed tied to the studied case. The side effect of this realism of context comes at the expense of the possibility for generalization [34].

III. IEC 62304 Medical device software certification

Once the prototype of MVM proved the feasibility of the project, the developers had to assure safety and usability in real environments. This started the effort to obtain the certification of the ventilator, including the software that should be compliant with the `IEC 62304` standard [16].

`IEC 62304` defines safety classes of the software, based on the potential to create an injury to the patient: Class A - no injury or damage to health is possible - Class B - non-serious injury is possible - and Class C - death or serious injury is possible. The standard itself prescribes a set of activities shown in Fig. 1, which must be performed (and documented) during the software development process. Based on the membership class, some of these activities are mandatory.

Regardless of the class the software belongs to, the standard requires the software development plan, as well as the software

[3]https://a16z.com/2019/08/16/software-eaten-world-healthcare/
[4]https://opensource.org/licenses/MIT

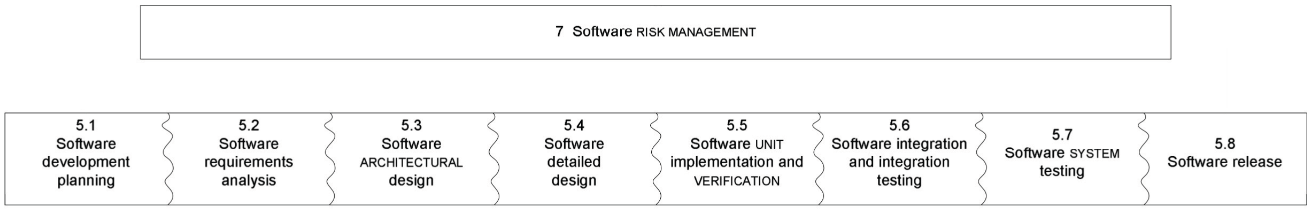
7 Software RISK MANAGEMENT
5.1 Software development planning
5.2 Software requirements analysis
5.3 Software ARCHITECTURAL design
5.4 Software detailed design
5.5 Software UNIT implementation and VERIFICATION
5.6 Software integration and integration testing
5.7 Software SYSTEM testing
5.8 Software release

Fig. 1: Main activities required by the `IEC 62304` standard

requirements specification documents. The former establishes the process followed during the development, the expected deliverables, and the software development life cycle model. The latter contains a description of the software functionalities, system inputs and outputs, interfaces between software and other systems, security requirements, user interface, etc. Once the software requirements are defined, the software architecture design is derived for software belonging to class B or class C. Furthermore, a software detailed design for each software unit is required for class C. For all classes, the standard requires performing software system testing of the entire system, e.g., verifying that for the input provided the expected output is obtained. This system testing activity corresponds to so-called integration testing in software engineering, and it is the only required testing for software in class A, while for software belonging to class B or class C, it comes after the (required) software unit verification at the component level, and software integration at the architectural level. The standard is flexible about organization of testing by types and test stage, but coverage of requirements, risk control, usability, and test types (e.g., fault, installation, stress) should be demonstrated and documented.

Finally, software release, software maintenance process, and software risk management process documents (the last only in case of class B and class C) are required for the certification.

Once the standard for software certification was identified, the MVM team started the effort to make the MVM development process and software compliant with the standard. This brought to the identification of a number of activities to be undertaken to assure the quality of the software embedded in MVM, and of a process of their organization able to combine the *rigidity* of the ISO standard development process with the *flexibility* of an agile attitude: the overall goal was to get the final certification in the fastest and most collaborative way but still having the rigor required by the standard. This *clear* and *dynamic* software development process brought to re-engineer most software components to provide evidence of a comprehensive and defensible argument that the system was acceptably safe to operate in the identified context.

IV. Development process

As already stated, we have undertaken several actions and activities in order to assure the quality of the MVM software according to the `IEC 62304` standard. The following sections present the process with all the performed activities.

A. Software development planning

First of all, we have classified the device based on the potential to cause injuries, as required by the `IEC 62304` standard. Starting from the safety classification and taking into account the activities required by the `IEC 62304` standard (which are summarized in Fig. 1), the whole MVM has been classified in class C since death or serious injury is possible. Considering the safety class and the mandatory activities, we have defined the development process we intended to adopt, depicted in Fig. 2. The list of activities is:

1) *Software development planning*, which regards the entire MVM and maps to activity 5.1 in the standard including all the supporting activities.
2) *System requirements* and *Software architectural design*, which refer to the MVM and define the desired components, after a *risk analysis* has been performed.
3) *Software requirements analysis* and *Software detailed design*, which refer to a single component previously identified during the architectural design phase.
4) *Software implementation* and *Unit testing* which are performed for every component of the MVM ventilator.
5) *Integration* and *Validation testing*, which is performed by integrating all the software components with the hardware and by testing the system as a whole.

For each phase of the process, we have defined the tasks to be performed and the deliverables.

As shown in Fig. 2, the adopted process model resembles a V-model, and we mapped all the activities required by the standard to process phases. For each software component, we decided to work iteratively (inner dark circle in Fig. 2) to guarantee the conformance among requirements, architecture, design, and implementation and we have integrated the V-model with agile practices to favor flexible responses to changes due to the need to develop the ventilator as fast as possible. In this way, we could parallelize the process in an agile like mode and foster a collaborative approach. Moreover, agile practices are considered by the outer circle in Fig. 2, after validation/system testing all the processes can be re-executed to integrate solutions of detected problems. In the end, we can say that we combined various models of software development processes, namely, V-model with agile practices and model-driven development (mostly for the state machine component described in Sec. V-E).

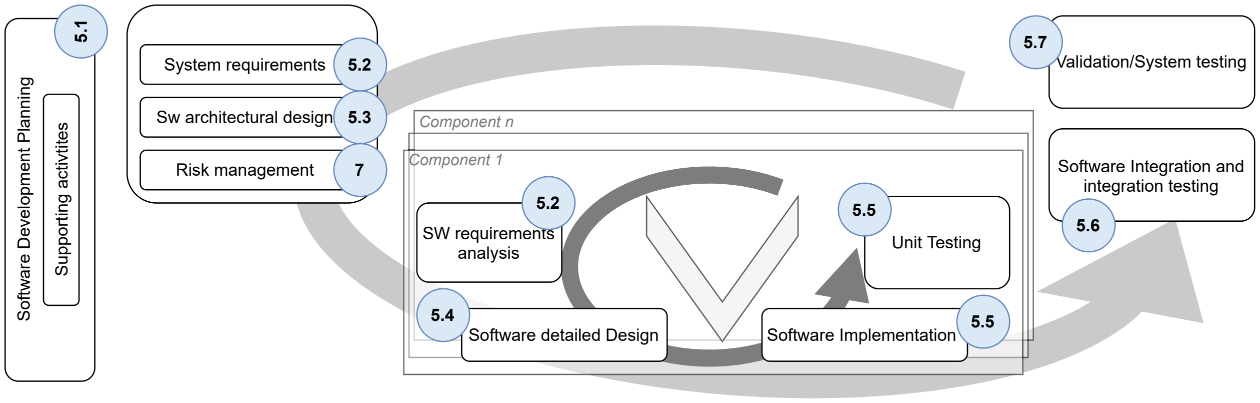

Fig. 2: MVM software development process

Lessons learned 1: 1) *IEC 62304 and V-Model:* The development process was strongly influenced by the IEC 62304 standard, so the V-model, although not mandated, is the "best fit" with regulatory requirements as it produces the necessary deliverables required when seeking regulatory approval. 2) *Use of agile practices:* However, it was necessary to integrate the V-model with agile practices, to combine efficiency, quality, maintainability, and flexibility.

B. Teams definition and meetings planning

The activities and the teams were organized as follows.

1) Teams definition: Seven subgroups were created ad hoc during the project taking into account competencies, workload, and availability of individuals. We identified a group leader, which was responsible for meeting deadlines and for reporting the progress to the project coordinator. Each activity was assigned to each team. The first group was in charge of defining the software development plan, supporting activities, and performing the risk analysis; the second group had the goal to define software requirements; the software architecture specification was designed by group three; group four developed the MVM software. Group five and six run software unit testing, and integration and system testing, respectively. Another group prepared the operational and maintenance manual.

2) Meetings planning: The whole MVM team met daily (including holidays) for 1 hour from 5 PM to 6 PM CET to check the status of the project and fix the goals for the next day. The project leaders went through the task list to check the status of each task and to check if some obstacles emerged. Moreover, each sub-team had to organize task-specific meetings to synchronize their work. Because of the lockdown due to the COVID-19 emergency in Italy, only a few in-person meetings were organized (and authorized by the special national commissioner). Moreover, since the teams include people from Europe and America, only the late afternoons and nights were usable for online meetings.

The MVM project was challenging also for what concerns the development team characteristics. The team was in fact (i) multidisciplinary and heterogeneous, involving people with different backgrounds including physicians, physicists, electronic engineers, and computer scientists, and (ii) composed of volunteer people motivated by the social nature of the project and by their passion.

Lessons learned 2: 1) *Coordination effort*: The project was successful but there was a quite huge overhead of coordination, with various calls every day from the morning till evening. The coordination of the team should not be underestimated. Open-source software development could be a good development experience from which projects of this nature can learn. 2) *Enlarging team*: We observed what is reported in the Mythical Man-Month [35]: adding people is not necessarily a good solution to improve the efficiency and effectiveness of a team. 3) *Commitment and participation*: Having responsibility for each sub-activity and setting strict intermediate goals have favoured commitment and participation.

C. Supporting activities

In parallel to the definition of the development process, we defined a set of supporting activities: project management, the definition of a change control process, the definition of the development environment, and the definition of code guidelines.

1) Tools and instruments for project management: Initially, we have selected the tools supporting us in project management. We had evaluated the use of a project management tool like Jira, but in the end, we decided to use only a combination of more accessible tools, considering the heterogeneity of the group, like Google Drive, GitHub, Zoom, and Slack. We have decided to adopt UML (Unified Modeling Language) to model system requirements and software architecture, and it turned out to be understandable also for people not experts in software engineering. We have put all the documents on Google Drive and the code on GitHub, which provides hosting for software development and version control using Git. This allowed people from all over the world to contribute to developing different parts of the software. This approach has allowed keeping track of all the changes in the code, which is really useful to manage and control the software development process. Furthermore, we have configured a continuous integration system using Travis CI[5] (Continuous Integration).

2) Reviewing process: For all the activities that foresee a document as output, we have applied two review steps. The first review was performed internally by one designated member of the team. Once the document was approved, the Design Authority (Elemaster - the company involved in MVM production) was in charge of reviewing such documents and producing the Review Acceptance document containing all the comments. Later, the comments were included in a new version which was resubmitted again to the Design Authority. This process has continued until the Design Authority has approved the final version of the document.

3) Document templates: The initial templates for the documents were kindly provided by the Canadian Nuclear Laboratories (CNL) sub-team, which has great experience in critical software certification (not in the medical field, though). This helped greatly to speed up the process from the start. To keep track of the links among requirements in the documents we have used an in-house tool developed by a CNL collaborator. This tool reads each document subject to traceability and generates a matrix of requirements. The tool has been daily executed to illustrate the links that exist and to report missing or faulty links.

Lessons learned 3: 1) *Multiplicity of tools:* The use of a great variety of tools (one tool for each particular purpose) even if not integrated and not specific for software project management, has provided indispensable support to the team. 2) *Templates and review process:* Having a partner that provided all the necessary templates and a clear review process has helped to define which activities should be performed. 3) *Use of UML:* Standard graphical notations like UML shown to improve communication and to be easily usable by non-software experts (very skilled in other fields, though).

V. Development phases

A. System requirements (5.2)

One of the obstacles met during the first implementation of MVM, was the lack of well-defined and traceable system requirements. This has caused a lot of misunderstanding between the developers of different components and some operations did not fit with the requirements of the standards that regulate the ventilator development. According to point 5.2 of the standard, the system requirements activity has been performed to define device requirements at a higher level of abstraction.

In this phase, we have defined functional, performance, safety, and cybersecurity requirements, the overall system structure, environmental conditions, materials, and human factors. Furthermore, each requirement has been uniquely identified by a number.

[5]https://travis-ci.org/

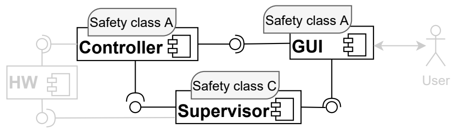

Fig. 3: Overall MVM software architecture

The major change discovered during writing software requirements was that the prototype was vulnerable to faults because, in case of failure of one of the components, the system was not able to put itself in a safe mode, such that the valves are positioned to allow the patient autonomous breathing. For this reason, the necessity to have a supervisor came out. This required a change of the initial electronic board with the introduction of a small micro-controller.

Reverse engineering some parts of the prototype was also useful in order to get a complete and consistent requirements specification of the device operation. It was applied when no enough information was available, and this was the case, for example, for the specification of the alarms. The reverse process also helped to reveal details useful to specify the duration of a trigger window, namely the time interval within which spontaneous breathing can be detected (in PCV mode).

Lessons learned 4: 1) *Written requirements:* Not having written requirements since the beginning led to having various attempts to address the requirements in different software components. For this reason, precise system requirements are very important also in an emergency situation. Having developers referring to the same written documents without inconsistencies reduces the development time. 2) *Reverse engineering:* For systems for which a prototype is present, especially if it is developed by domain experts, reverse engineering has shown to be a viable solution for discovering functionalities and configuration parameters to be included in system requirements. 3) *Need of a traceability system:* A traceability system helps developers to trace all the requirements and their changes through all the development process.

B. Software Architecture Design (5.3)

As shown in Fig. 3, the architecture was designed to be composed of three main components (or units according to the terminology of the standard), namely Graphical User Interface (GUI), Control Software (Controller), and Supervisor Software (Supervisor).

The GUI is the software running on the touch screen panel. It displays information to the user like airway pressure, minute volume, positive-end expiratory pressure for the most recent breath, respiratory rate, and peak inspiratory pressure. Furthermore, it acquires data from the user: ventilation settings and alarm settings. The controller receives user inputs from the GUI, e.g. the start/stop ventilation command. It implements the state machine of the ventilator behavior, which has mainly

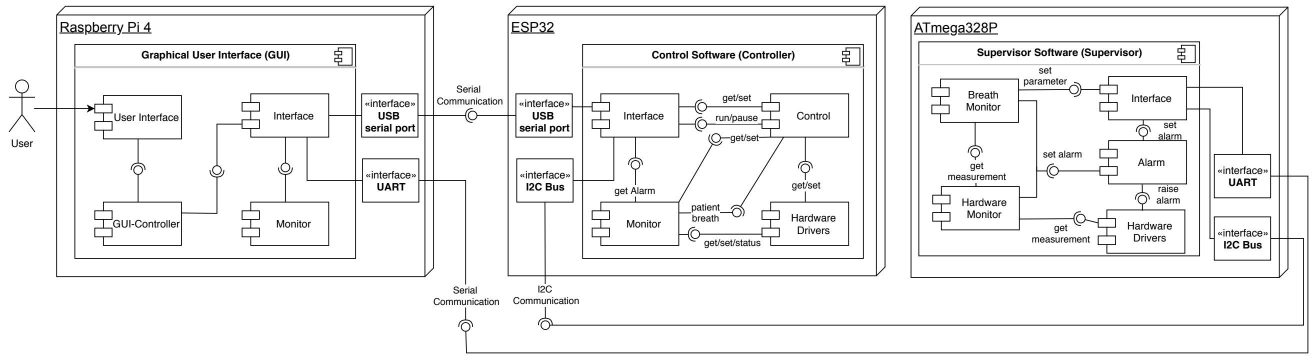

Fig. 4: Detailed MVM software architecture

three phases: ventilation off, running in PCV mode, and running in PSV mode. Based on the current state, it opens/closes the input and output valves. Moreover, the controller manages ventilator alarms in case of errors. The supervisor monitors the overall behavior of the system. It checks if the controller, the GUI, and the hardware are working as expected, and, in case of errors, it raises alarms. Furthermore, the supervisor forces the machine into a safe mode to prevent patient injuries in case of errors during ventilation.

Lessons learned 5: 1) *Upfront aspects balancing:* As we can learn from software architecting, it is important to go towards "just-in-time architecture" [33] and to find a balance between upfront aspects (what is planned before the start of development) and emerging aspects (what appears as decisions are taken in the course of the development, e.g. by fixing wrong assumptions or making decision deliberately postponed) [36]. 2) *Importance of the architecture:* Software architecture is still important even during emergency development. In fact, we have experimented that without a well-defined architecture (as for the prototype), it was not clear how software components were supposed to synchronize and exchange information among them.

C. Risk management (7)

In the risk management activity, each component of the MVM software has been classified based on the potential to create injuries (see safety classes in Fig. 3) as required by the `IEC 62304` standard. Both GUI and controller are classified as class A software because their behavior, despite affecting the operation of the machine, does not cause patient injuries. On the other hand, the supervisor is the most critical component, since it both forces the machine into a safe mode to prevent patient injuries in case of errors during ventilation and intervenes in case of GUI and controller failures.

Lessons learned 6: 1) *Safety assurance effort:* Isolating safety-critical features, by organizing the system in different components, has allowed us to focus the safety assurance effort on a limited portion of the system.

D. Software Requirement Analysis (5.2) & Software detailed design (5.4)

The third activity consists of specifying for each software component the requirements in a separate document, detailing those introduced in the system requirements. The documents contain functional and capability requirements, software inputs and outputs, interfaces between software and other components, alarms and warnings, user interface requirements, and requirements related to system installation and maintenance. We have widely used state machines to define the behavior of GUI, Controller, and Supervisor. Often, we have used diagrams and drawings, and in few cases a more powerful tool, such as Yakindu (more in Sect. V-E), which has been used in order to identify both states and events that trigger the change of state. For each state, we have defined the detailed behavior, the user inputs, the expected outputs, performance, and failure conditions. Furthermore, for each software component, we have defined the software unit interfaces, to ensure that the software subsystem will communicate properly with external components. Fig. 4 refines Fig. 3 and shows the detailed architecture of MVM.

Users interact with the User Interface subcomponent, and the interaction is mediated by the GUI controller and the Interface component, which manages the connection with the control software - via a USB serial port - or with the Supervisor - via a UART interface. The monitor component is used to monitor the interaction with the GUI.

The logic of the Control Software (Controller) is in the Control subcomponent. This component is in turn composed of two subcomponents, the valve controller controlling the valves and the state machine component. The state machine component controls the operation modes (i.e. ventilation off, running in PCV mode, and running in PSV mode). The controller receives user input from the GUI, e.g. the start/stop ventilation command. The vital signs of the patient are monitored by the monitor subcomponent, and it manages ventilator alarms in case of errors. The controller also interacts with the hardware through the Hardware Drivers component to open or close the input and output valves.

The Supervisor Software (Supervisor) component gets mea-

Software unit	Lines of code	
	Prototype	Released version
GUI	14,347	26,027
Controller	4,653	14,331
Supervisor	NA	2,689

TABLE II: Lines of code of the MVM software units

surements from the Hardware Drivers component, monitors both the hardware devices and the patient breath, and, when needed, e.g. when switching to the safety mode, raises alarms and changes parameters of the controller and the GUI.

Lessons learned 7: 1) *Modularity and parallelization:* Designing a product in a modular way has been a successful decision, since, in a distributed project (such as the one of the MVM) it has allowed different teams to work in parallel on different parts of the system. 2) *State machines for wide interpretability:* We have found that using state machines, for the specification and the design, has contributed to favor the discussion on the adopted solutions even with people not used to software development since graphical representations are easily understandable.

E. Implementation (5.5)

Some software components of the existing MVM prototype needed to be re-engineered and re-implemented. Table II reports the lines of code of the three main software units (before the re-engineering effort and at the end).

The *GUI* component is written using the PyQT5 framework in Python[6]. We have added several new functionalities expected by the software requirement specifications. For example, many alarms have been changed, in terms of thresholds and behavior. Moreover, to avoid problems in the communication between GUI and controller, we have devised a new communication protocol used by the GUI to send/receive messages to/from the controller. In particular, we had to implement in the communication protocol the guidelines defined in the `IEC 61784` standard [37].

The MVM *controller* is written in C++ and it has been changed a lot with respect to the prototype. One of the parts that have been completely rewritten is the *state machine*. We have introduced the use of the Yakindu Statechart tool[7] for its implementation. Fig. 5 reports a high-level version of the state machine for the controller, showing the states in which the ventilator can be and the events that cause state changes. In detail, after the startup and the self-test phases, the machine is in the ventilation-off mode. From that, it can go either to the PSV or PCV modes. Inside these modes, there are other sub-states (including inspiration and expiration) not reported in the figure to keep it simple and readable. The C++ code of the controller state machine has been automatically generated from the Yakindu model.

[6]Qt is set of libraries for accessing many aspects of modern desktop and mobile systems. PyQt5 is a set of Python bindings for Qt v5. More info at: https://pypi.org/project/PyQt5/

[7]https://www.itemis.com/en/yakindu/state-machine/

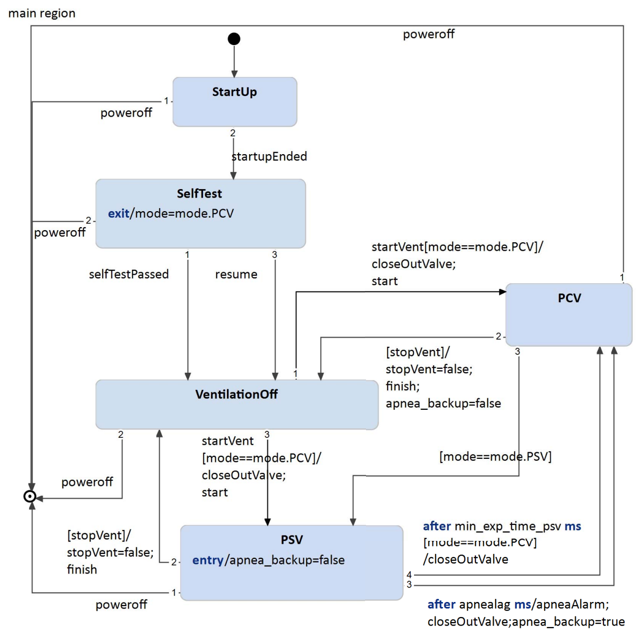

Fig. 5: Yakindu state machine of the MVM

Since the re-engineering process has involved not only the software but also the hardware part of the MVM, e.g. the valve controller has been changed too, to fit with the HW modifications. The change led to the definition of a new tuning method. The controller has been also adapted in terms of *alarms* since they have to comply with the ones that have been added or modified in the GUI code. In particular, all the alarms have been implemented to comply with the `IEC 60601-1-8` [38] and `ISO 80601-2-12` [39] standards.

In the re-engineered MVM version, the supervisor has been developed in C++ from scratch, and it required the addition of two different software serial lines: one used to communicate with the GUI and one with the controller.

Lessons learned 8: 1) *Mix of programming languages:* Using several programming languages in a single project is usually discouraged [40]. However, in an emergency (such as during COVID-19) in which the products have to be delivered as soon as possible, we have experienced that having more languages allows the inclusion of more developers and speeds up the implementation process, with only a minimal effort in integration of the code. 2) *Coding standards and guidelines:* Sharing the coding standards and guidelines (e.g., the importance of comments [41]) with all the people involved in the implementation phase is of key importance, in particular with heterogeneous development groups, even during emergency development. 3) *Advantages of state machines in implementation:* State machines added flexibility and maintainability since it was very simple to modify it and then regenerate code, which was directly integrated, through a wrapper, to the hand-written code.

F. Unit Testing (5.5)

The testing activities have been executed in parallel to the implementation, since every test failure has required corrections in the code, and it has been performed against the unit software requirement specifications.

As MVM has different components, each one implemented using a different programming language, we have used several testing frameworks. We have unit-tested the *GUI* using PyTest-Qt[8], which has allowed us to simulate users by faking clicks on the buttons. To test the *controller* and the *supervisor*, we needed to mock the hardware to emulate the interaction with it, and test the software units by using the Catch2 framework[9] and the Trompeloeil mocking library[10].

Besides, we have configured a continuous integration system on Travis CI to guarantee that new software implementations did not compromise the functioning of the already existing code. By using the CI tool, we have re-executed all the unit tests for every commit made on each component.

We have tracked all the testing failures using the GitHub issues tracking system and this has allowed us to monitor the progress of the fixes in the software.

Lessons learned 9: 1) *Testing not only safety-critical components:* Defining in advance the safety classes of all the components in the developed system can increase significantly the rapidness of testing activities. In fact, medical software safety standards do not mandate extensive unit testing for class A components. Thus, a good practice is to design the system in a modular way, isolating all the non-dangerous functions (i.e., in class A) that testers can limitedly check. 2) *Importance of testing:* Besides what is required by the standards, testing activities are important when performed for safety-critical components. This is a consolidated aspect when working with software engineers but not for all the people composing heterogeneous teams such as the MVM ones. 3) *Advantages of CI tools in community projects:* As MVM has been a community project, where a lot of people have worked at the same time on the same system, CI tools have proved to be crucial for maintaining under control the modifications made by all the developers.

G. Integration (5.6) & Validation Testing (5.7)

This activity has required incremental integration between software units and hardware components. The main challenge has been the need of having the hardware available since some of the software components included in the controller or supervisor require direct interaction with it. Even if during the first integration phases the HW can be ignored, it must be considered for the final integration steps. Thus, final integration testing phases have been performed on-site, only by the people working in the company that produces the MVM, or by the ones that had a physical version of the ventilator. As we worked with a ventilator, we had to simulate the patients. For this purpose, we have used an active lung simulator, such as the `ASL 5000` [42], or a passive mechanical one. After the execution of the integration testing activities, for each one of the test cases, we have reported the outcomes in Integration Test Procedure and Integration Test Report documents.

During validation testing the whole system has been tested as a unit, to confirm the correct behavior of MVM according to the system requirements. This activity has been performed over an actual physical version of MVM and simulating the patient breath by using the `ASL 5000` active lung simulator. It has been guided by the `ISO 80601-2-12` standard [39], to prove the basic performance and usability of the ventilator.

Lessons learned 10: 1) *Integration testing for SIMDs:* It is particularly challenging to develop and validate software-in-medical-devices (SIMDs) and in general, systems that integrate hardware, software, and mechanics by distributed teams. Often, real hardware is needed for testing the software that is affected or affecting a piece of hardware. Software-in-the-loop simulation is often a good solution to this challenge; however, it is not a solution in general. In fact, simulation requires a special setting with professional simulation tools and an accurate hardware model. They are not always easily available, especially in a context in which the hardware is under development as the software is. Moreover, as we can learn from robotics, "*Current simulation solutions are not able to emulate real-world phenomena in a sufficiently realistic manner.* " [43].

VI. Related Works

There are lately several research projects that study the impact of the COVID-19 pandemic on the way software developers work [44]–[47]. These papers try to relate practices adopted because of the pandemic, especially Working From Home (WFH), with the productivity and the well-being of software developers working on the usual software projects. In all these papers, the authors signal a decrease in productivity due to the difficulty of coordination between colleagues and to the stress because of the pandemics. In our MVM experience, we cannot say the same: people worked harder because of the need and willingness to finish as soon as possible a crucial product for COVID-19 patients. Not having direct access to the physical devices of course has had a negative impact on some activities but the team found several solutions to overcome also these problems (e.g., see Sect. V-G).

On the other hand, there are studies and experience reports about using already existing software, and IT approaches, for an emergency, also applied to COVID-19 patients [48], [49]. However, no particular attention is given to software development. Other studies are focused on developing new medical software but in a classical software development environment [50], [51]. However, our study is different from that, since it is addressing the problem of developing and certifying a piece of software *for* an emergency *amid* the emergency itself. Some suggestions like the integration of agile

[8]https://pypi.org/project/pytest-qt/
[9]https://github.com/catchorg/Catch2
[10]https://github.com/rollbear/trompeloeil

practices and the need for rigorous requirement specification have been applied to our project as well, but we expect that medical SW development without any emergency is more planned and structured, and some of the lessons we learned may not hold.

Other studies describe the development of new software and AI solutions for COVID-19 during the emergency. The area of medical imaging is, as expected, the most relevant [52]. Machine Learning algorithms have been promptly adapted to the diagnosis of COVID-19 cases. In [53], the authors present a rapid AI development cycle for an automated detection solution using deep learning CT Image Analysis. Those papers focus more on data collection and management than software development. Regarding, instead, the SW development of medical devices for COVID-19, we can compare MVM with the numerous open-source community-driven projects working on mechanical ventilators [54]. We could not found reports on their experience in SW development and certification. Not surprisingly, only a few of them[11] reached the certification while many others, even if very promising, are still behind. We believe that this paper could provide a guideline for them as it would help, God forbid, future similar projects.

VII. Conclusion and Final remarks

In this paper, we have presented our experience regarding the development and certification of the MVM, a mechanical ventilator for COVID-19, with respect to the `IEC 62304` standard. The certification required a re-engineering of the entire software since in the prototype the software component was underestimated. What we learned from this experience is that obtaining certification is not just producing the documentation that is needed to get a stamp from a certification authority, but instead, we had evidence of the value of being obliged to follow a certification process even in emergency. Besides having an apparently functioning prototype, the effort of adhering to a certification process led to discovery various malfunctions and errors in the implementation. For instance, before the re-engineering, there was no clear maximal limit on the duration of alarm audio pause, while its duration shall not exceed 120 sec without healthcare professional operator intervention. Moreover, in the implementation, there was no clear distinction between snooze (to allow the operator to pause the active auditory alarm signals) and acknowledge (to allow the operator to cease the alarm signal for which no associated alarm condition currently exists - alarm reset). The standard requires snooze and acknowledge to be two separate functionalities.

In only 42 days from the initial prototype production to the demonstration of performances (the fastest approval by the FDA starting from a concept), the FDA (Food and Drug Administration) declared that the MVM falls within the scope of the Emergency Use Authorization (EUA) for ventilators[12]. The EUA is released when certain safety criteria are met even without the formal SW certification. Moreover, after we have completed the re-engineering activity described in this paper, the certification request has been forwarded to the Health Canada agency and to European Certification Authority to get the CE marking (following the standard certification process).

At the end of September 2020, MVM obtained the Health Canada Authorization[13] and at the beginning of May 2021 the CE marking. Thank to these achievements, the MVM can be now sold and used not only in the USA but also in Canada and Europe. Many countries all around the world have manifested interest in MVM ventilators. There is an ongoing project which aims to deliver MVM devices where they are most needed, and it is currently being sold by an African Union charity[14]. At the moment the ventilator is produced by Vexos, which has started the first production batch of 10,000 pieces to be delivered to the Government of Canada, and Elemaster, which has actively contributed in the project.

In this paper, we shared the lessons learned about the development of the MVM under hard constraints. Tab. III summarizes the lessons learned and maps them to the research questions. We believe that the experience and the lessons learned we report in this paper would help other groups working on similar projects, requiring certification by organizations such as FDA, Health Canada, and CE.

RQs	**Lessons Learned (LL)**
RQ1 (*Process*)	LL1.1 - `IEC 62304` and V-Model
	LL1.2 - Use of agile practices
	LL3.1 - Multiplicity of tools
	LL3.2 - Templates and review process
RQ2 (*Activities*)	LL4.1 - Written requirements
	LL4.2 - Reverse engineering
	LL4.3 - Need of a traceability system
	LL5.1 - Upfront aspects balancing
	LL5.2 - Importance of the architecture
	LL6.1 - Safety assurance effort
	LL7.1 - Modularity and parallelization
	LL8.1 - Mix of programming languages
	LL8.2 - Coding standards and guidelines
	LL8.3 - Advantages of state machines in implementation
	LL9.1 - Testing not only safety-critical components
	LL9.3 - Advantages of CI tools in community projects
	LL10.1 - Integration testing for SIMDs
RQ3 (*People*)	LL1.2 - Use of agile practices
	LL2.1 - Coordination effort
	LL2.2 - Enlarging team
	LL2.3 - Commitment and participation
	LL3.3 - Use of UML
	LL7.1 - Modularity and parallelization
	LL7.2 - State machines for wide interpretability
	LL8.1 - Mix of programming languages
	LL8.2 - Coding standards and guidelines
	LL9.2 - Importance of testing

TABLE III: Mapping between RQs and lessons learned

Acknowledgment

The authors would like to thank the entire team who contributed to the realization of the MVM project.

[11]https://bit.ly/2POPm3g
[12]https://bit.ly/3dcZ6vs
[13]https://bit.ly/30K3CfX
[14]https://breathoflifeafrica.org/#MentorProject

References

[1] A. Kornecki and J. Zalewski, "Software certification for safety-critical systems: A status report," in *2008 International Multiconference on Computer Science and Information Technology*, 2008, pp. 665–672.

[2] G. Ferreira, C. Kästner, J. Sunshine, S. Apel, and W. L. Scherlis, "Design dimensions for software certification: A grounded analysis," *CoRR*, vol. abs/1905.09760, 2019.

[3] A. Gannous and A. Andrews, "Integrating safety certification into model-based testing of safety-critical systems," in *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*, 2019, pp. 250–260.

[4] A. Wölfl, N. Siegmund, S. Apel, H. Kosch, J. Krautlager, and G. Weber-Urbina, "Generating qualifiable avionics software: An experience report (e)," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2015, pp. 726–736.

[5] A. Wölfl, "Data management in certified avionics systems," Ph.D. dissertation, Universität Passau, 2018.

[6] A. Hovsepyan, D. Van Landuyt, S. Op de beeck, S. Michiels, W. Joosen, G. Rangel, J. Fernandez Briones, and J. Depauw, "Model-driven software development of safety-critical avionics systems: an experience report," vol. 1249. Hebig, Regina, 2014, pp. 28–37.

[7] R. Pietrantuono and S. Russo, "Robotics software engineering and certification: Issues and challenges," in *2018 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, 2018, pp. 308–312.

[8] J. L. de la Vara, E. Parra, L. Alonso, R. Mendieta, B. López, and J. M. Álvarez-Rodríguez, "Integration of tool support for assurance and certification and for knowledge-centric systems engineering," in *2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, 2019, pp. 326–329.

[9] N. Hrgarek, "Certification and regulatory challenges in medical device software development," in *2012 4th International Workshop on Software Engineering in Health Care (SEHC)*, 2012, pp. 40–43.

[10] S. Pelayo, S. Bras Da Costa, N. Leroy, S. Loiseau, and M.-C. Beuscart-Zéphir, "Software as a medical device: Regulatory critical issues," *Studies in health technology and informatics*, vol. 183, pp. 337–42, 02 2013.

[11] I. Lee, G. J. Pappas, R. Cleaveland, J. Hatcliff, B. H. Krogh, P. Lee, H. Rubin, and L. Sha, "High-confidence medical device software and systems," *Computer*, vol. 39, no. 4, pp. 33–38, 2006.

[12] M. N. K. Boulos, A. C. Brewer, C. Karimkhani, D. B. Buller, and R. P. Dellavalle, "Mobile medical and health apps: state of the art, concerns, regulatory control and certification," *Online Journal of Public Health Informatics*, vol. 5, no. 3, feb 2014.

[13] J. Neto, J. Damásio, P. Monthaler, and M. Morais, "Product-based safety certification for medical devices embedded software," *Studies in health technology and informatics*, vol. 216, pp. 227–31, 08 2015.

[14] W. J. Gordon and A. D. Stern, "Challenges and opportunities in software-driven medical devices," *Nature Biomedical Engineering*, vol. 3, no. 7, pp. 493–497, jul 2019.

[15] A. Ohne Autor Fd, "General Principles of Software Validation; Final Guidance for Industry and FDA Staff, Version 2.0," FDA document formal, Jan. 2002. [Online]. Available: http://www.fda.gov/MedicalDevices/DeviceRegulationandGuidance/GuidanceDocuments/ucm085281.htm

[16] *IEC 62304 Medical device software — Software life cycle processes*, International Electrotechnical Commission Std.

[17] "On the role of formal methods in software certification: An experience report," *Electronic Notes in Theoretical Computer Science*, vol. 238, no. 4, pp. 3 – 9, 2009, proceedings of the First Workshop on Certification of Safety-Critical Software Controlled Systems (SafeCert 2008).

[18] A. Bombarda, S. Bonfanti, and A. Gargantini, "Developing medical devices from abstract state machines to embedded systems: A smart pill box case study," in *Software Technology: Methods and Tools*. Springer International Publishing, 2019, pp. 89–103.

[19] P. Arcaini, S. Bonfanti, A. Gargantini, A. Mashkoor, and E. Riccobene, "Integrating formal methods into medical software development: The ASM approach," *Science of Computer Programming*, vol. 158, pp. 148–167, jun 2018.

[20] S. Russo and F. Scippacercola, "Model-based software engineering and certification: Some open issues," in *2016 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, 2016, pp. 237–240.

[21] R. A. Schrenker, "Software engineering for future healthcare and clinical systems," *Computer*, vol. 39, no. 4, pp. 26–32, 2006.

[22] N. Lalband and K. Dwaram, "Software engineering for smart healthcare," vol. 8, pp. 325–331, 04 2019.

[23] J. H. Weber-Jahnke, M. Price, and J. Williams, "Software engineering in health care: Is it really different? and how to gain impact," in *2013 5th International Workshop on Software Engineering in Health Care (SEHC)*, 2013, pp. 1–4.

[24] E. Nazari, M. Shahriari, M. Edalati, and H. Tabesh, "Create frameworks from software engineering to health care: A survey article info abstract," *Journal of Biostatistics and Epidemiology*, 01 2019.

[25] M. McHugh, F. McCaffery, and V. Casey, "Barriers to adopting agile practices when developing medical device software," in *International Conference on Software Process Improvement and Capability Determination*. Springer, 2012, pp. 141–147.

[26] M. McHugh, F. McCaffery, and G. Coady, "An agile implementation within a medical device software organisation," in *Software Process Improvement and Capability Determination*, A. Mitasiunas, T. Rout, R. V. O'Connor, and A. Dorling, Eds. Cham: Springer International Publishing, 2014, pp. 190–201.

[27] M. Mc Hugh, O. Cawley, F. McCaffcry, I. Richardson, and X. Wang, "An agile v-model for medical device software development to overcome the challenges with plan-driven software development lifecycles," in *2013 5th International Workshop on Software Engineering in Health Care (SEHC)*. IEEE, 2013, pp. 12–19.

[28] M. C. Di Guardo, E. Marku, W. M. Bonivento, M. Castriotta, F. Ferroni, C. Galbiati, G. Gorini, and M. Loi, "When nothing is certain, anything is possible: open innovation and lean approach at mvm," *R&D Management*, vol. n/a, no. n/a. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1111/radm.12453

[29] P. Runeson and M. Höst, "Guidelines for conducting and reporting case study research in software engineering," *Empirical software engineering*, vol. 14, no. 2, p. 131, 2009.

[30] A. Abba *et al.*, "The novel mechanical ventilator milano for the COVID-19 pandemic," *Physics of Fluids*, vol. 33, no. 3, p. 037122, mar 2021.

[31] R. N. Westhorpe and C. Ball, "The manley ventilator," *Anaesthesia and intensive care*, vol. 40, no. 5, pp. 749–750, 2012.

[32] M. Andreessen, "Why software is eating the world," *Wall Street Journal*, vol. 20, no. 2011, p. C2, 2011.

[33] P. Pelliccione, E. Knauss, R. Heldal, S. Magnus Ågren, P. Mallozzi, A. Alminger, and D. Borgentun, "Automotive architecture framework: The experience of volvo cars," *Journal of Systems Architecture*, vol. 77, pp. 83 – 100, 2017. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1383762117300954

[34] K.-J. Stol and B. Fitzgerald, "The ABC of software engineering research," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 27, no. 3, p. 11, 2018.

[35] F. P. Brooks, *The Mythical Man Month*. Prentice Hall, 1995. [Online]. Available: https://www.ebook.de/de/product/3236893/frederick_p_brooks_the_mythical_man_month.html

[36] R. Wohlrab, U. Eliasson, P. Pelliccione, and R. Heldal, "Improving the consistency and usefulness of architecture descriptions: Guidelines for architects," in *2019 IEEE International Conference on Software Architecture (ICSA)*, 2019, pp. 151–160.

[37] *IEC 61784 - Industrial communication networks*, International Electrotechnical Commission Std.

[38] J. Edworthy and C. Baldwin, "Medical audible alarms and IEC 60601-1-8," *Proceedings of the Human Factors and Ergonomics Society Annual Meeting*, vol. 60, no. 1, pp. 634–635, sep 2016.

[39] International Organization for Standardization. (2020) ISO 80601-2-12:2020. [Online]. Available: https://www.iso.org/standard/72069.html

[40] P. Mayer, M. Kirsch, and M. A. Le, "On multi-language software development, cross-language links and accompanying tools: a survey of professional software developers," *Journal of Software Engineering Research and Development*, vol. 5, no. 1, apr 2017.

[41] J. Raskin, "Comments are more important than code: The thorough use of internal documentation is one of the most-overlooked ways of improving software quality and speeding implementation." *Queue*, vol. 3, no. 2, pp. 64–65, 2005.

[42] A. Dexter, N. McNinch, D. Kaznoch, and T. A. Volsko, "Validating lung models using the ASL 5000 breathing simulator," *Simulation in Healthcare: The Journal of the Society for Simulation in Healthcare*, vol. 13, no. 2, pp. 117–123, apr 2018.

[43] S. Garcia, D. Struber, D. Brugali, T. Berger, and P. Pelliccione, "An empirical assessment of robotics software engineering," in *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2020)*, 2020.

[44] D. Ford, M.-A. Storey, T. Zimmermann, C. Bird, S. Jaffe, C. Maddila, J. L. Butler, B. Houck, and N. Nagappan, "A tale of two cities: Software developers working from home during the COVID-19 pandemic," 2020.

[45] P. Ralph, S. Baltes, G. Adisaputri, R. Torkar, V. Kovalenko, M. Kalinowski, N. Novielli, S. Yoo, X. Devroey, X. Tan, M. Zhou, B. Turhan, R. Hoda, H. Hata, G. Robles, A. M. Fard, and R. Alkadhi, "Pandemic programming," *Empirical Software Engineering*, vol. 25, no. 6, pp. 4927–4961, sep 2020.

[46] D. Russo, P. P. Hanel, S. Altnickel, and N. van Berkel, "The daily life of software engineers during the covid-19 pandemic," 2021.

[47] C. NicCanna, M. A. Razzak, J. Noll, and S. Beecham, "Globally distributed development during covid-19," 2021.

[48] B. A. Jr, "Use of telemedicine and virtual care for remote treatment in response to covid-19 pandemic," *J. Medical Syst*, vol. 44, no. 7, p. 132, 2020.

[49] A. Asadzadeh, S. Pakkhoo, M. M. Saeidabad, H. Khezri, and R. Ferdousi, "Information technology in emergency management of covid-19 outbreak," *Informatics in Medicine Unlocked*, vol. 21, p. 100475, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2352914820306262

[50] C. Denger, R. L. Feldmann, M. Host, C. Lindholm, and F. Shull, "A snapshot of the state of practice in software development for medical devices," in *First International Symposium on Empirical Software Engineering and Measurement (ESEM 2007)*, 2007, pp. 485–487.

[51] M. McHugh, O. Cawley, F. McCaffcry, I. Richardson, and X. Wang, "An agile V-model for medical device software development to overcome the challenges with plan-driven software development lifecycles," in *2013 5th International Workshop on Software Engineering in Health Care (SEHC)*, 2013, pp. 12–19.

[52] F. Shi, J. Wang, J. Shi, Z. Wu, Q. Wang, Z. Tang, K. He, Y. Shi, and D. Shen, "review of artificial intelligence techniques in imaging data acquisition, segmentation, and diagnosis for covid-19," *IEEE Reviews in Biomedical Engineering*.

[53] O. Gozes, M. Frid-Adar, H. Greenspan, P. D. Browning, H. Zhang, W. Ji, A. Bernheim, and E. Siegel, "Rapid ai development cycle for the coronavirus (covid-19) pandemic: Initial results for automated detection & patient monitoring using deep learning ct image analysis," 2020.

[54] J. M. Pearce, "A review of open source ventilators for COVID-19 and future pandemics," *F1000Research*, vol. 9, p. 218, apr 2020.