© 2016 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

Quality Assurance in Agile Safety-Critical Systems Development

Tom McBride
Faculty of Engineering and IT
University of Technology Sydney
Australia
tom.mcbride@uts.edu.au

Marion Lepmets
Regulated Software Research Centre
Dundalk Institute of Technology
Dundalk, Ireland
marion.lepmets@gmail.com

Abstract **– In this position paper we examine how safety could be assured when increasingly complex systems are developed using agile software development methods. We first discuss the source and nature of complexity in software systems and how a probe – sense – learn approach recommended by the Cynefin Framework is appropriate for designing complex systems and a sense – analyse – learn approach is appropriate for developing a complicated system whose design has been determined. We then examine how quality assurance is incorporated into agile software development before pointing out that those characteristics of a self-managed team that produce so many benefits for software development of complex systems whose solution evolves with problem understanding, are also vulnerable to confirmation bias. This suggests that for safety critical system development, software systems developed by agile teams will need verification and validation by independent parties. We review current quality management practices for medical device software development before discussing how our earlier findings could be adopted into safety critical software quality management.**

Keywords – Safety critical, software development, complex systems, quality assurance

I. INTRODUCTION

Safety-critical systems are becoming increasingly complex as the number of their components, the interactions between the components and the interactions with the wider sociotechnical systems in which they are embedded all increase [1, 2]. In such an environment safety is a characteristic that emerges from the components themselves and interactions between the different parts of the system. Safety-critical systems can also fail in complex and unforeseeable ways due to the incompatibilities between various components as well as their interactions. To manage such increasing complexity during both development and operation, the Cynefin framework [3, 4] suggests using a probe-sense-learn approach suitable for complex domains. In software development, such an approach best corresponds to agile methods' iterative development and rapid customer feedback loops, employing self-organizing teams of highly skilled individuals. Despite various benefits of agile methods, they don't yet incorporate the quality assurance practices central to safety-critical systems development. In this paper, we argue that some quality assurance practices should be incorporated into agile software development of safety-critical systems to increase reliability and safety of such systems. The paper is organized as follows: we first describe the complexity of safety-critical systems with the example of medical device domain. We then provide an overview of the Cynefin framework practices that are best suited for the complex domain and illustrate the point through the uptake of agile methods in safety-critical software development today, their benefits as well as their shortcomings. We discuss quality assurance in agile development methodologies before discussing how quality management for development of safety critical devices, medical devices in particular, might change in order to take advantage of the benefits of agile development without compromising the high levels of assurance.

II. COMPLEX SYSTEMS

Safety-critical systems are growing in complexity and becoming more innovative, with software being the main source of both their complexity and their innovations. Medical devices used to be primarily electrical and mechanical systems with very little, if any, software. This has changed drastically as software systems are now essential components of medical device control functions. In fact, from 2007, medical devices could be solely made up of software if they satisfied the medical device definition and operated on a general purpose IT platforms intended to be placed on the market without a dedicated hardware. An example is an iPad application that advises doctors on the correct drug dosage amounts for a patient [5]. Additional examples include the bionic ear, first switched on for a commercial patient in 1982, and the heart pacemaker that has been steadily enhanced since the 1980s. More advanced bionic devices are under development and hold

great promise of significant improvement in their recipient's quality of life. Medical diagnostic devices too are advancing technologically, bringing with them promises of great health benefits. However, technologically advanced devices are often complex and, while acknowledging their benefits, they also come with the disadvantages of complex systems. Even if the medical device was simple, it is used within a socio-technical system that is itself complex. Try as we might to anticipate, to plan, to smooth things out, complex systems have global behaviours that emerge unpredictably from the interaction of its components [6].

The majority of innovations in medical devices today are made to its software component and, as with all complex systems, they can behave in unexpected ways under some conditions [7]. Since there is the potential for harm, safety-critical devices must gain regulatory approval before being released for use. Often the auditors, who check for conformance to all of the relevant regulations, may not be able to discover critical errors prior to market release, resulting in an increasing number of device recalls. In 2011, 24% of all medical device recalls in the US (including devices with no embedded software) were attributable to software errors alone and the trend is growing rapidly [8].

III. INNOVATION AND FAILURES

Creating something novel brings with it a risk that the result will be other than what was intended. A measure of that risk can be taken from the repeated empirical finding that the risk of a new product failure is about 40% [9]. That is, the product was not a commercial success or did not achieve its objectives in one way or another. This is not to say that the product itself was faulty, just that it was not a commercial success. In some ways this can be taken as a measure of our inability to predict the outcomes of the complex system of creating a new product. It is reasonable to theorise that our ability to predict the behaviour of a complex software system will be about the same and that there will be a 40% unintended and potentially harmful effects in any new system. Despite our best efforts to understand completely our designs and the systems we develop, we are unable to predict how that system will behave in the complex environment of its deployment.

An immediate and obvious response to such rates of failure would be to increase the amount of oversight in the processes of innovation. This has been the case for medical devices with an increase in regulation following the Therac-25 failure [10]. However, increased oversight, in the form of regulation and quality management systems, impede an organization's ability to respond to market demand or competitive changes [11, 12]. In addition, the growing number of product recalls indicates that the current regulations, and the theories on which they are based, might not be the best that can be done. Essentially the problem is to determine how to detect and control risks of harm arising from innovations in complex medical devices. To explore this question we will first examine how the Cynefin framework distinguishes between complex situations and complicated situations, and how it provides strategies for dealing with each. Then we will show that software development involves both complex and complicated processes, and the consequences of each. We will explore what these different consequences mean for quality assurance in software development. Finally we will discuss quality assurance for complex medical devices before drawing some conclusions.

IV. CYNEFIN AND SOFTWARE DEVELOPMENT

The Cynefin Framework (Figure 1) offers a way to make sense of different types of systems and situations, providing both symptoms and appropriate responses. Although there are five distinguishable states in the framework, for this paper we will discuss only two; the complicated and the complex.

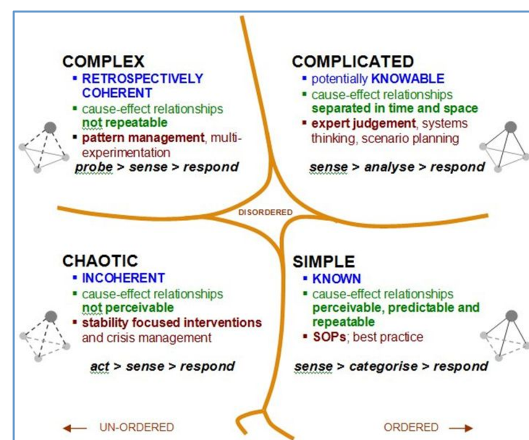

Figure 1: Cynefin Framework

When the situation is complicated the most appropriate response is sense – analyse – respond. The situation is knowable, although it may take the knowledge of several experts to determine just how to sense the situation and then to analyse that information in order to know how best to respond. In software development an example of complicated process might be configuration management. In this process there could be numerous relationships between the configurable items but it would be possible to determine those relationships and the state of the overall configuration item repository. Similarly the process used to build a software system from its various components may be complicated but here too there is a determinate answer. More subtle perhaps is the process of software construction in which the designed software is converted into executable code. Set aside for a moment the possibility that a programming language did not support a necessary construct or algorithm. In the absence of such a

problem, realising a design into the final product is a complicated problem but one in which the realised product can be verified against its design. Defects arising during, or from, a complicated processes are defects of production. That is, the defect has arisen from a faulty performance of the process. This approach seems to have been favoured in software development with its emphasis on requirements elicitation and analysis, and quality management methods taken from manufacturing production. Such an approach can work well when the situation is knowable.

When the problem is complex, predicting a correct answer, or even what would produce a correct answer, becomes difficult if not impossible. This situation suits the Cynefin suggested probe – sense – learn approach in which something is tried that can be used to learn more about both the problem and possible solutions. An answer is correct in hindsight. For example, software development planning usually requires an estimate of how long a task will take. Despite years of experience the outcomes of estimation remain uncertain because no-one can say for certain what interruptions they may get, what may prove more difficult to do, what might not work as the estimator thought it would, and a number of other reasons. When the task is done it is then possible to say exactly how long it took and precisely why it took that long.

Rather than software development being simply a matter of implementing a fully understood solution to a problem, software developers now participate in the co-evolution of both the business problem and its solution. A business problem may have several different acceptable solutions depending on the available technology and what can be achieved within the applicable constraints. At the start of development there will be some projections of what system components must be developed, and how they must be developed, in order to deliver the required features and qualities. Invariably there are some parts of the system that won't work as we thought they should work, or the functionality provided by some component does not work the way we thought it worked. Whatever the reason our predictions are not realised and it is only after completion are we able to say exactly how the system was built and why it was necessary to build it that way. At this point it is useful to distinguish between software design and software production. For the purposes of this argument software design includes all of the activities that discover, determine or decide what software constructs are necessary to provide the functions and features that will provide an acceptable solution to the problem. Software production includes all of the activities that realise that design in the form of an executable software system. Design can occur at a high level, when the overall architecture of the system is decided, at the low level, when the implementation details of an algorithm are decided, and all points in between. The point of distinguishing between software design and software production is that each requires a different approach and different forms of quality control. Software design is complex. The outcome is difficult to predict but will make sense in hindsight. An appropriate strategy for software design is probe – sense – learn that is achieved quite well through incremental development so typical of agile software development methods. Software production, converting a completed software design into executable software, can be complicated but is not complex. The eventual state of the executable software is entirely predictable even if the means of production is difficult and may require the combined efforts and expertise of several experts. Software production suits a sense – analyse – respond approach where past production is sensed, the resulting data analysed and used to modify and improve the production methodology. However, software development seldom happens in a two-step process where the system is first designed then produced, but in a single pass in which one approach dominates and the other is accommodated as best it can be. There are exceptions to this, the most obvious being when there is some form of prototype or proof of concept developed before the main development.

Software development has more of the characteristics of a design problem than a production problem; complex and fully understandable only in hindsight.

The terms "science" and "design" used to label the different approaches to problem solving could be applied to this discussion. Complicated problems and situations seem to suit a science approach, based on deduction and induction, because they are knowable. Complex problems do not suit a science approach because the problem is, to some degree, unknowable. Lawson [13] found that science graduates tended to solve problems by first understanding their structure before venturing an answer whereas architecture graduates tended to explore the problem by generating a number of solutions as a way of exploring the problem until one of the solutions proved acceptable. Several explanations of how designers work have proposed similar explanations – that designers use abduction to generate a possible solution, then evaluate the solution against its constraints and requirements rather than use induction or deduction [see e.g.14, 15]. The systems analysis based approach of 40 years ago has all the characteristics of fully exploring the problem until it is understood well enough that the best solution can be formulated, in the form of specifications for a system to be developed. Agile software development, in all its various forms, uses the design approach by producing a small part of the system and evaluating its acceptability, then using what has been learnt about the problem to develop more of the system. The agile approach explores the problem by

producing and testing possible solutions. Such an approach carries with it the risk that a seemingly feasible solution is later revealed to be feasible only for a subset of possible situations. In design terms, it was a local optimum and the solution search stopped prematurely.

Software systems themselves can be complex and adaptive. During development the design and implementation of different software components can change to adjust to one or more components with which they interact. For example, there may be a business requirement of X, say, which could be satisfied by system functionality Y. However, on investigation or during system architecture design or even later in the development life cycle it is discovered that system functionality Y cannot be delivered or will not satisfy X. Possibly business requirement X might now be modified, along with all other business requirements impacted by this change in X. In turn this has a flow on effect to system functionality Z. Another example is when a component might interface to a database sub-system or a protocol stack for which there are vendor supplied interface components. If the vendor changes those interface components then the effects of those changes propagate through the system under construction, possibly back to the conception of the business problem itself. The eventual states of software elements are retrospectively coherent, whether requirements, design or code. That is, only after completion it is possible to say exactly how they happened and why they finished up that way, but not before completion.

V. Quality Management in agile development

Agile software development, in its various forms, has been described in books, blogs, conference papers, journal articles and trade press. The essential characteristics of the various forms of agile software development are those that enable a problem to be explored by generating potential solutions and evaluating those solutions in order to discover more about the problem. Additionally agile teams are usually small, self-managing and comprised of a majority of experienced developers. They are small enough that they can coordinate their work using informal mutual adjustment rather than the more formal methods required of larger teams. Self-management is enabled by their knowledge and experience in software development, and is necessitated by the variability of the work at any stage. Self-management also enables the team to respond quickly to changing situations. Their knowledge and experience with software development means that most of the time agile teams do not need much supervision and don't need to be supported by formal software development or quality management processes. Instead, these teams, for the most part, are able to determine for themselves what objectives they should pursue, and to devise a process for that specific pursuit. Importantly, teams seem to decide how their work should be verified and validated.

Largely missing from the innumerable discussions about agile software development is acknowledgement of the trade-offs that have been made to enable the speed and responsiveness of agile development versus the rigour of plan-based development. There is a fondness for citing the "Agile Manifesto" without acknowledging that the people who developed it were all very experienced and skilled developers. They didn't need the support or constraints of formal software development methods but had, instead, acquired and would apply very sound professional practices that achieved all of the same objectives that a formal development method would do. However, being very experienced and with sound professional practices, they were able to vary their development method to suit the circumstances. Developing a proof of concept does not need rigorous quality management and it doesn't matter if the resulting system is riddled with bugs so long as it proves or disproves the concept. The full rigour of a formal development method simply hinders innovation and responsiveness [12]. At the other end of the scale, using less than the full rigour of a formal development method when developing a large critical system is likely to produce a lower quality system. After all it was to achieve such high quality levels that rigorous methods were developed. Between those two extremes of very fast responsive development that depends on very experienced software developers and very careful development that can support inexperienced developers lies the broad range of agile development methods in which experienced professionals can adjust their development methods to suit the circumstances. Alistair Cockburn tries to formalise such a range in his Crystal Methodologies [16].

The verification and validation practices are of most interest for this discussion. Agile advocates do encourage some forms of independent check. That could be a peer review, pair programming or some variation. Importantly, that review is by a peer, someone capable of understanding what needs to be done, what has been done and why. A second common form of validation is unit testing, also normally performed by the developers themselves. But most attention is concentrated on the developer's perspective and interests, not a quality assurance perspective. For example, texts describing the various forms of agile development do not dwell on what happens once the software has been developed nor on the forms of independent verification and validation. Although agile development can support very high standards of software quality there is no guarantee that this will always happen.

An important difference between agile development teams and the more traditional project staffing is that the agile team is expected to be

sufficiently skilled that all tasks can be completed by the team alone whereas the traditional team continues the long established industrial practice of skill specialisation. For a traditional project, within a functional organization work flows through different specialisations or, in a matrix organization, different specialisations are called upon at different times during the project. While this increases the coordination overhead and can slow a project down it does mean that, potentially, the best available knowledge and skills can be used.

Agile teams depend on their collective knowledge and skills being good enough for the project. The agile approach seems to have provided acceptable quality for a lot of projects but there is very little empirical information available on quality control practices and outcomes of agile teams compared to traditional teams. On the surface of it, a team of specialists should be able to achieve higher quality related outcomes than a team of generalists. Yet domains other than software development also must achieve high performance in complex situations.

VI. High Performance Teams

In a study into high performance teams such as you might find in surgery, nuclear power plant operation, chemical processing and flight deck operations, Weick *et al* [17] identify several characteristic behaviours of highly reliable organizations. Weick *et al* list these as;

- preoccupation with failure,
- reluctance to simplify,
- sensitivity to operations,
- commitment to resilience, and
- deference to expertise

These characteristics may also occur in many software development teams but how could their practice be demonstrated? How could they be audited? A preoccupation with failure could be demonstrated through a "Definition of Done," a checklist of work to be completed before the work can be considered potentially shippable [18]. Such a checklist and its successful completion would qualify as an auditable record. Potentially, a definition of done is also a place where lessons learnt from past failures could be recorded.

Reluctance to simplify things and a sensitivity to operations are also present in agile teams when the team includes or seeks input from end users and problem owners. While this might not always be true, it is far more possible than when developers' understanding of what needs to be done is separated from operations by layers of specifications and organizational distance.

One of the characteristics of high performance teams that might be hard to audit is their commitment to resilience, which shows through their learning from failures, their abilities to detect, contain, and bounce back from errors that are part of an indeterminate world [19]. In software development such resilience could be evident through a record of process review and improvement, multiple reviews of work in progress, multiple opportunities to detect and correct faults in the software. Within agile development is the practice of holding a retrospective. Retrospectives provide opportunities to consider what worked well, what failed, what could be improved and, more importantly, what should be done to affect those improvements. Like most software development practices, can be performed well or poorly but whether done well or not they provide a means to learn from failures, to achieve some level of resilience.

The final characteristic of high reliable organizations is a deference to expertise, which requires diversity to help detect more in a complex environment and helps to do more with the complexities they find. One of the differentiating characteristics of agile software development is that roles are not specialized. Instead, team members exercise their specific expertise when it is required but otherwise do whatever is necessary on the day.

In complex, changeable, critical, time-dependant situations high performance teams are characterised by some behaviours that could usefully be adopted by agile development teams. There is, however, a critical difference between the high performance teams studied by Weick *et al* and teams developing critical software systems: software development is not so time critical that decisions cannot be reviewed.

VII. Confirmation Bias

Although the structure, culture and practices of agile software development can demonstrate many of the characteristics of highly reliable organizations, they have their weaknesses. As Weick notes, a subtle trap that many of those who need to manage the unexpected is that they fail to describe the important mistakes people should guard against in pursuit of their objectives [19: p9]. The agile development practice of allowing development within the core team to be verified and validated by the team itself is vulnerable to confirmation bias. That is, a developer and a development team knows what the software is intended to do so will tend to conduct tests and seek evidence to confirm that the software does what is intended. This is not negligence or incompetence by the development team but simple seeking evidence or interpreting evidence according to an established belief [20]. Khaneman [21] argues that "the operations of associative memory contribute to confirmation bias." Philosophers of science may advise testing hypotheses by deliberately trying to refute them but most others, Kahneman argues, will tend to seek data that are compatible with the beliefs they currently hold. Turner [22] cites a number of cases

where entire organizations held beliefs that were later contradicted with fatal consequences. One that was especially tragic was the belief that safety in coal mines concerned what happened underground and that anything above ground was not a concern. Mine spoil was tipped on a hill above a village but the dangers of doing so were not regarded as a matter of mine safety. After many years and in very wet conditions the spoil slid down the hill in the form of a slurry to bury parts of the village beneath. A software system may be intended for a specific purpose but that does not mean it will achieve its intentions, or that the purpose will remain constant or that it will be used for that purpose alone.

To overcome confirmation bias, to effect a truly independent audit of the developed software, the developer would need to document what their requirements were, what decisions they made about implementing those requirements and the rationale for doing so, and probably need to document the code itself just so that it could all be found and understood by someone not already familiar with the development. This introduces some formality, reduces development speed and responsiveness. Suddenly it starts looking like formal process-based development.

To some degree test driven development (TDD) [23] does try to overcome the tendency for a developer to believe their software does what they intend it should. However a developer is unlikely to understand how a user might use the system or the full circumstances in which the system could be used so is unlikely to construct tests that fully exercise the system. Any number of test specialists and authorities (e.g. James Bach, Cem Kaner, Michael Bolton, Rex Black) would argue that a critical system is unlikely to be tested as well by a developer as it would be by a specialist tester.

Safety critical software, indeed all critical software, seems to require some design practices and apply some design constraints, require some development practices and apply some development constraints. Since a major reason of adopting agile software development methods is to support problem and solution co-evolution, and a process approach to quality management would compromise the essential innovation and responsiveness, ways of achieving high quality of and during design requires a different approach. The high performance team approach described by Weick *et al* provides a promising way forward on this. However, even the most complex and novel software must be realised into executable code at some stage and that is where the more familiar process based approach to software quality can be applied.

VIII. QUALITY MANAGEMENT FOR MEDICAL DEVICE SOFTWARE.

Quality management for medical devices is dominated by the regulations and guidance published by European Commission [5, 24-26] and the US Food and Drug Administration [27]. An explicit FDA guidance on general principles of software validation [28] takes a process approach to quality assurance in which a process uses resources to transform inputs into outcomes. Of central importance to the FDA guidance are the processes for verification and validation which are harmonised with ISO 8402:1994 and which treats “verification” and “validation” as separate and distinct terms. Verification “provides objective evidence that the design outputs of a particular phase of the software development life cycle meet all of the specified requirements for that phase” whereas validation seeks “confirmation by examination and provision of objective evidence that software specifications conform to user needs and intended uses, and that the particular requirements implemented through software can be consistently fulfilled.” The guidance dismisses verification in a single paragraph but devotes more than 25 pages to validation, describing the principles, the activities and tasks of validation. These are modelled on the validation process of ISO 15288 [29] where processes are described in a way that assumes a flow of work and control from inputs to outputs. That is, the input to the process is specified at the beginning and it is up to the process to deliver outputs that reflect those inputs. The processes of ISO 15288, and ISO 12207 that preceded it, reflect the needs of large organizations to define processes in a way that supports contractual distributed development. Potentially each process could be contracted to a different supplier. When work is distributed and governed by contracts it makes sense to validate process outputs against their requirements and to verify that the process activities have been performed effectively. This type of quality management assumes that the whole will be a sum of its parts and that correct functioning of the parts will assure correct functioning of the whole. But a complex system produces emergent outcomes from interactions among the parts. Even though the parts may function correctly, the overall system may function incorrectly.

It is difficult to anticipate all possible combinations of interactions between parts of a complex system and, as Perrow [30] points out, accidents arise from unexpected combinations. This becomes a problem of determining where accidents are likely, then testing to see that the system prevents or avoids such accidents. To date software testers have adopted several strategies to maximise testing return for effort. Among the strategies are risk based testing and scenario testing which, in different ways, consider possible circumstances in which a software system failure would have significant consequences. Since people have a bias toward confirming what they believe to be true (subjective dissonance), efforts to think of possible failures tend not to include those circumstances of combinations believed unlikely or impossible [31].

Industries other than those developing medical devices have this same problem of assuring safety within the unpredictability of a complex system and they share some common characteristics, as previously discussed. In high performance organizations safety is not assured through compliance, although compliance may contribute toward safety, but through preoccupation with failure, reluctance to simplify, sensitivity to operations, commitment to resilience and deference to expertise [17].

IX. Discussion

Agile software development combines problem-solving, design and software production. It is the iterative nature of agile software development that supports gradual understanding, evolving and refining an acceptable solution. Within the problem solving cycle there is design of potential solutions or design of refinements to existing systems. Designs are usually evaluated against guidelines, constraints and other knowledge about feasible designs. Although the software development community has many guidelines, few have been formalised into any quality management system. Design rules, coding standards, object-oriented design guidelines are freely available and easily tailored to a specific organization or situation, and they are the easiest place to manage knowledge about software development. Industry leading companies are adopting some very advanced design guidelines. An example is the O-O object design SOLID principles and specializations [32]. As systems become larger and more complex, it becomes important to apply these design principles more rigorously. Agile software development methods seem increasingly to be adopting a "Definition of Done" to ensure all those actions that should be done before the software can be considered potentially shippable have been completed. The "Definition of Done" provides a checklist that can be updated to reflect lessons learnt from previous failures.

Designs must be realized by being converted into a functioning system while retaining fidelity with the design. Such realization is closer to production than it is to design and this is where the accumulated experience with quality assurance can be applied. This is where it is much more possible to plan, to use established processes with much greater confidence of achieving the expected result. Again this can be expressed only as being more confident of the result because much of the detailed design of a software system is done during coding, especially during agile development. To fully separate design from the much more predictable realization of that design would require re-establishing the habit of performing detailed design before and separate from coding itself. This inserts formality into the development process, bringing it closer to the specification and plan-based development and seems unnecessary since a certain amount of design during implementation seems to be manageable without everything becoming unpredictable. It seems a matter of degree that many organizations have managed to establish, given the number who claim to be using agile methods, for what otherwise seem to be routine development projects.

With a complex system it is not only the integrity of a component that matters but also interactions between components. The separate components of a system could be correctly implemented but the design itself may be flawed. The Therac-25 failure was a failure of design, not of implementation. While there are a number of ways to check the correctness and integrity of a program, the ways of checking a design for correctness or, in this case, safety, are less well known. With a complex system it becomes impossible to prove that the system will be safe because an unsafe outcome could arise from a combination of events, usually an unexpected combination [30]. Rather than demand that all possible combinations of events be tested it would be better to follow the example of high reliability organizations that have already adopted a mindset and behaviours that guide the organization to prevent unsafe outcomes.

Software developers individually have produced some rules or guidance about what works well and what to avoid. At the most abstract are 'patterns' and for which the Portland Pattern Repository is probably the most well-known. Object-oriented design has thrown up some guidelines for well-designed objects; the SOLID guidelines. Most programming languages have at least one set of coding rules, often with a tool to evaluate some code for breaches of the rules. Rules or guidance, and compliance to them, won't assure safety simply because the situation can throw up new outcomes faster than the rules can be developed to prevent them. Instead safety must be assured through behaviours that use but do not stop at the available rules and guidance.

X. Conclusion

Medical device software system development will inevitably adopt an agile approach in order to deal with complex problems and situations. Agile software development offers safety critical developers more than the usual advantages of self-management and responsiveness during problem discovery. Team composition and many of the agile practices coincide quite well with practices found in highly reliable organizations so can be expected to higher quality, more resilient software systems.

However, agile software development teams do not necessarily always perform or support those practices. Critically agile development teams, without some review of their development by a party external to the development team itself, are vulnerable to confirmation bias since they know how the software system is intended to work and are highly unlikely to think of how it could fail to

work. If they could, they would have prevented that potential failure during development anyway. So any claim that the developed software system is safe would normally require some external demonstration that this is so. This would lead to some formal communication between the development team and the verification and validation team.

Agile software development methods support problem-solving and design much better than traditional plan-based methods but so far it seems that methods to review software system design have remained informal, relying on the professional skills of the developers and not on formal methods of validation. Design faults usually are more subtle and of more consequence than production faults so developers of software systems for critical systems, medical device systems in particular, should consider how software and system designs might be validated.

XI. References

[1] M. Jordon, H. J. Lanham, R. A. Anderson, and R. R. McDaniel Jr, "Implications of complex adaptive systems theory for interpreting research about health care organizations," *Journal of evaluation in clinical practice,* vol. 16, pp. 228-231, 2010.

[2] D. F. Sittig and H. Singh, "A new sociotechnical model for studying health information technology in complex adaptive healthcare systems," *Quality and Safety in Health Care,* vol. 19, pp. i68-i74, 2010.

[3] H. W. Dettmer. (2011, Systems Thinking and the Cynefin Framework. Available: http://goo.gl/uamHDB

[4] C. F. Kurtz and D. J. Snowden, "The new dynamics of strategy: Sense-making in a complex and complicated world," *IBM Systems Journal,* vol. 42, 2003.

[5] European Commission:2007 Directive 2007/47/EC of the European Parliament and of the Council concerning medical devices

[6] J. Pelrine, "On Understanding Software Agility" A Social Complexity Point of View," *Emergence: Complexity & Organization,* vol. 13, pp. 26 - 37, 17 Sep 2011 2011.

[7] N. Leveson, "Medical Devices: The Therac-25," in *Safeware: System Safety and Computers*, ed: Addison-Wesley, 1995, pp. 1-49.

[8] FDA. (2012, 12.04). *FDA News on Software Failures Responsible for 24% of all Medical Device Recalls*. Available: http://www.fdanews.com/newsletter/article?articleId=147391&issueId=15890

[9] G. Castellion and S. K. Markham, "Perspective: New Product Failure Rates: Influence of Argumentum ad Populum and Self-Interest," *Journal of Product Innovation Management,* vol. 30, pp. 976-979, 2013.

[10] N. Leveson, "Medical devices: The therac-25," *Appendix of: Safeware: System Safety and Computers,* 1995.

[11] M. J. Benner. (2007, January 2009). Dynamic or Static Capabilities?: Process Management Practices and Response to Technological Change. Available: http://goo.gl/sO7VSn

[12] M. J. Benner and M. Tushman, "Process Management and Technological Innovation: A Longitudinal Study of the Photography and Paint Industries," *Administrative Science Quarterly,* vol. 47, pp. 676-706, 12 2002.

[13] B. R. Lawson, "Cognitive Strategies in Architectural Design," *Ergonomics,* vol. 22, pp. 59-68, 1979/01/01 1979.

[14] N. Cross, "Designerly Ways of Knowing," in *Designerly Ways of Knowing*, ed London: Springer London, 2006, pp. 1-13.

[15] K. Dorst, *Understanding design*, 2 ed.: BIS, 2006.

[16] A. Cockburn. (2004, July 21 2004). *Crystal Methodologies*. Available: http://alistair.cockburn.us/crystal/crystal.html

[17] K. E. Weick and K. M. Sutcliffe, *Managing the unexpected: Assuring high performance in an age of complexity*. San Francisco: Jossey-Bass, 2001.

[18] K. S. Rubin, *Essential Scrum: A Practical Guide to the Most Popular Agile Process*: Addison-Wesley, 2013.

[19] K. Weick, "Managing the Unexpected: Complexity as Distributed Sensemaking," in *Uncertainty and Surprise in Complex Systems*. vol. 4, R. McDaniel and D. Driebe, Eds., ed: Springer Berlin / Heidelberg, 2005, pp. 51-65.

[20] R. S. Nickerson, "Confirmation bias: A ubiquitous phenomenon in many guises," *Review of general psychology,* vol. 2, p. 175, 1998.

[21] D. Kahneman, *Thinking, Fast and Slow*: Penguin, 2011.

[22] B. A. Turner and N. F. Pidgeon, *Man-Made Disasters*, 2nd ed. Oxford: Wykeham Publications, 1997.

[23] K. Beck, *Test-driven development: By example*. Boston: Addison-Wesley, 2003.

[24] European Commission:1990 Council directive 90/385/EEC on active implantable medical devices (AIMDD)

[25] European Commission:1993 Directive 93/42/EEC of the European Parliament and of the Council concerning medical devices

[26] European Commission:1998 Directive 98/79/EC of the european parliament and of the council of 27 october 1998 on in vitro diagnostic medical devices

[27] *General Principles of Software Validation; Final Guidance for Industry and FDA Staff,* US Department of Health and Human Services, 2002.

[28] FDA:2002 FDA's General Principles of Software Validation; Final Guidance for Industry and FDA Staff

[29] ISO/IEC 15288:2008 Systems engineering - System life cycle processes

[30] C. Perrow, *Normal Accidents: Living with High-Risk Technologies*. Princeton: Princeton University Press, 1999.

[31] M. Syed, "Black Box Thinking: The Surprising Truth About Success," 2015.

[32] G. Suryanarayana, G. Samarthyam, and T. Sharma, *Refactoring for Software Design Smells: Managing Technical Debt* 1st ed.: Morgan Kaufmann, 2014.