AQI | Automotive Quality Institute

Challenges for quality management in an agile environment

Project report

Project no.	9810011
Project duration	January 01, 2018 – December 31, 2018
Autor(s)	Gunnar Harde (Automotive Quality Institute GmbH)
Date	January 26, 2019
Email	kontakt@aqigmbh.de

This publication is based on the expertise of AQI and its scientific partners. It represents a consolidated position on the topic under examination.

Content

Preface

The automotive industry is currently undergoing a transformation towards greater digitalisation of its products and services. This change affects not only the type of products but also the type of product development and maintenance: While previously phase-oriented product development processes dominated the automotive industry, agile approaches, such as those successfully used in software development for many years, are becoming more important. This increasing agility, which is becoming apparent in the product lifecycle of software products in the automotive industry, also calls into question the previous understanding of quality management and requires new answers from quality management in the transition to agile development methods in the automotive industry.

This report of the project “Challenges for quality management in an agile environment” deals precisely with these tasks facing quality management. The report consists of four parts:

The first part, “Agility”, gives an overview of the topic of agility. The focus of this part is agile software development, because agility has its origin and is established there. However, many of the principles and methods described there can also be meaningfully applied to development beyond software development. The possibilities and limits of agility are described, along with the basic ideas for scaling agility in large development projects.

The second part, “Agility and Software Quality”, deals with quality management in agile software development. The effects of agile values and principles on the quality of the software and how the quality management changes due to the changed development approach are described. Techniques used in agile development for test and delivery automation today allow very short development and delivery iterations and thus change the understanding and cooperation of development, operation and quality management.

In the third part, “Softwaredevelopment in the automotive industry”, the most important standards for the development of software in the automotive industry are considered and examined with regard to their compatibility with agility. The standards considered include Automotive SPICE, ISO 26262, AUTOSAR and the Maturity Level Assurance for New Parts according to VDA.

The fourth part, “QM of agile software development in the automotive industry”, gives recommendations for quality management of agile-developed products in the automotive industry. This section takes into account the previously described topics of agility, software quality and industry-specific standards and describes their effects on quality management in the automotive industry.

1. AGILITY

Since the formulation of the *Manifesto for Agile Software Development*, the development of software according to the agile approach has become more and more established and is now indispensable in many companies and departments where software is developed. Meanwhile, more and more organisations that do not develop software are discovering agility for themselves: in IT operations, in the development of hardware, in reorganisation projects and even in agile management of entire companies.

It is because of the current great popularity of agility that the understanding of what is meant by agility has suffered. Often keywords from the agile scene have been adopted without reflection and without really having understood the ideas behind them. This usually doesn't do any good for the organisations concerned or the agile approach, which is often justified in its use but also has its limits. An approach characterised by sobriety and a search for clarity is appropriate.

Agility can be applied meaningfully in many but not all areas. This part of the project report explains the term *agility* and shows what prerequisites are necessary for real agility – for the development of software and possibly beyond.

1.1. Definition of terms

Agile – it sounds like mental and physical mobility and vitality ("Grandma is still quite agile"). Everyone knows the colloquial term *agile* and has positive associations with it. This complicates the understanding of *agility*[1] as it is meant in the *Manifesto for Agile Software Development* (Beck et al. 2001a; hereinafter simply referred to as the *Agile Manifesto*): A self-organised team of experts develops software in close and continuous cooperation with the client on an incremental-iterative and result-oriented basis.

The Agile Manifesto fundamentally describes the way in which software should be developed from the perspective of the authors. The fact that the authors used the term *agile* is plausible, since

[1] It is common to use the term *Agile*. Dave Thomas, a co-author of *Agile Manifesto*, advocates the use of the term *agility* (Thomas 2015) instead. The author agrees.

they expressly demanded a rejection of the usual, strongly phase-oriented and often heavyweight software development approaches to date. They expressly welcome changes even at a late stage of development. Such organisational mobility is quite similar in nature to the colloquial use in the sense of mental and physical mobility, but in contrast to this, like all management approaches, has side effects and does not make sense in every context.

First, a few definitions:

The Agile Manifesto defines *agile values* (Beck et al. 2001a) and twelve *agile principles* behind these values (Beck et al. 2001b). The Agile Manifesto defines what is meant by agility for software development. The values are:

"We are uncovering better ways of developing software by doing it and helping others do it.
Through this work we have come to value:

Individuals and interactions over processes and tools

Working software over comprehensive documentation

Customer collaboration over contract negotiation

Responding to change over following a plan

That is, while there is value in the items on the right, we value the items on the left more."

Ken Schwaber and Mike Beedle have also identified *Scrum values* as a prerequisite for agile change and agile work: self-commitment, focus, openness, respect and courage (Schwaber/Beedle 2002). These values are important prerequisites for agility but are also relevant to many other operational and non-operational areas. They therefore do not define agility.

Agile practices are specific best practices that are often used in agile developments. These include pair programming, user stories to gather requirements, planning poker for effort estimation and test-driven development. These practices are not part of the Agile Manifesto but help to implement some of the principles.

Finally, the *agile methods* (also: *agile frameworks*) are specific guidelines for implementation of agility. They consolidate and align agile practices. Methods are different, partly in competition and partly complementary. Agile methods are, for example, Kanban, Scrum and Extreme Programming. Agile methods, like agile practices, are not part of the Agile Manifesto but more or less explicitly refer to it. Agile methods and practices are therefore concrete forms of agility but are not absolutely necessary for agile work.

The values and principles of the Agile Manifesto are the yardstick for assessing whether software is developed agilely or not. It is not so much a matter of implementing the individual principles to the letter, but rather that all the agile principles of the Agile Manifesto are lived out in their essence. This means:

- A team in which the manager determines how the tasks are to be carried out and delegates them to individual employees does not work agilely.

- A project in which concepts are presented to the customer as partial results long after the start of the project is not agile.
- An assignment based on a legally binding and unchangeable specification cannot be implemented in an agile manner.

1.2. The essence of agility

Agility essentially consists of two parts: the incremental-iterative development of the software and the self-organisation of the team.

1.2.1. Incremental-iterative procedure

The incremental-iterative development of software has a decisive advantage over conventional phase-oriented software development: The client can recognise their requirements much better on the basis of what has been created so far, concretely specify them and – as long as they provide the financial resources – steer them into development. They do not have to imagine an end product in detail in advance and specify it completely in order to find out in the end that it does not meet expectations and actual needs. The less clear the business requirements and technical conditions are, the greater the advantage of such an approach. Especially with complex products, the team often achieves much better solutions with the incremental-iterative approach than by elaborate preliminary specification. Hence the two principles of the Agile Manifesto:

"**Responding to change** over following a plan"

"**Customer collaboration** over contract negotiation"

Instead of specifying requirements in advance as the basis of the contract, the customer should work closely with the contractor and incorporate their requirements into the ongoing development process.

The incremental-iterative approach is not limited to the identification of requirements and their steering into development but also includes the development itself: software is tested and integrated in short iterations, and development processes are regularly evaluated, questioned and improved in teams.

Dave Thomas therefore describes agility as follows (Thomas 2015):

- "Find out where you are
- Take a small step towards your goal

- Adjust your understanding based on what you learned
- Repeat”

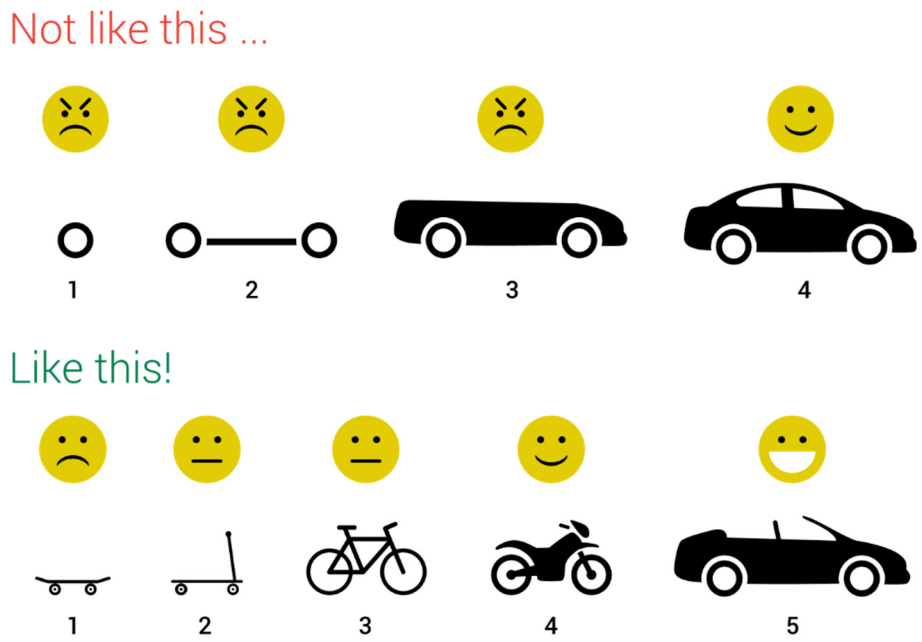

Figure 1: Incremental-agile product development

1.2.2. Self-organisation

Conventional project organisation is like playing table football: There are offensive players, midfielders and defenders who are attached to poles (= role description) and the player outside the playing field (= project leader) who decides who moves and kicks when. This Taylorist approach of fragmentation of activities according to functional criteria and their control by a superordinate management is to be overcome by self-organisation of the team: There are still offensive players, midfielders and defenders, but they move freely on the pitch and take responsibility together as a team. Self-organisation means playing football and not table football.

Software development is complex. In a complex environment, there can be no general rules. Rules can only be context-specific. The development team knows this context much better than management, which only sporadically has to deal with development directly.

Externally defined processes, including the tools to be used for them, limit the team’s scope for creative freedom and thus limit self-organisation. In contrast, the Agile Manifesto emphasises the importance of team members and their skills and internal cooperation:

“**Individuals and interactions** over processes and tools”

In addition, the success of a team should be assessed by the result to be achieved, and the result is the software to be developed. Apart from the system documentation for later maintenance of the software and the user manuals (if required), all other documents are intermediate results. If the focus is on this documentation, these intermediate results are in focus and the actual goal is increasingly lost sight of. Hence the Agile Manifesto:

"**Working software** over comprehensive documentation"

It should be the team that decides what it wants to document and to what depth. This also includes their responsibility to ensure long-term maintainability.

1.2.3. Incremental-iterative procedure and self-organisation

How are incremental-iterative procedures and self-organisation connected? A team can organise itself even if it works through a serial plan, that is, it does not approach the target incrementally-iteratively. Incremental-iterative development is neither a prerequisite nor a characteristic for self-organisation.

Conversely, an incremental-iterative approach is only efficient if the team organises itself. The team sharpens the complex task iteratively and develops the software incrementally. To this end, the team should be able to contribute all its knowledge quickly and in close coordination and should not have to wait for decisions from outside or above. This is exactly what self-organisation is for.

1.3. Employee motivation

The Taylorist imperative is: break down a process into the smallest possible work steps, assign these work steps to individually specialised but not necessarily highly qualified employees, provide detailed instructions on how to carry out the work steps, and monitor the work of the employees. The appropriate motivational principle is: punish misconduct and unfulfilled promises and reward good performance with praise, money and career prospects.

A number of psychological studies have shown that such a carrot-and-stick method actually works, assuming that the work is extremely simple and routine. However, the studies also show that as soon as a task requires a little creativity or analytical ability, this attempt at providing motivation does not work (described in detail in Pink 2009). On the contrary, it has been shown that performance in demanding tasks tends to decline when money is offered as a reward.

Other factors are crucial for the motivation of employees: **autonomy** in the exercise of work, **mastery** in the profession practised and the **purpose** that employees see in their work.

Software development is a complex and demanding task. Motivation through praise, reprimand and monetary incentives works just as little for software development as it does for most post-industrial work. The salary is a hygiene factor, not motivation.

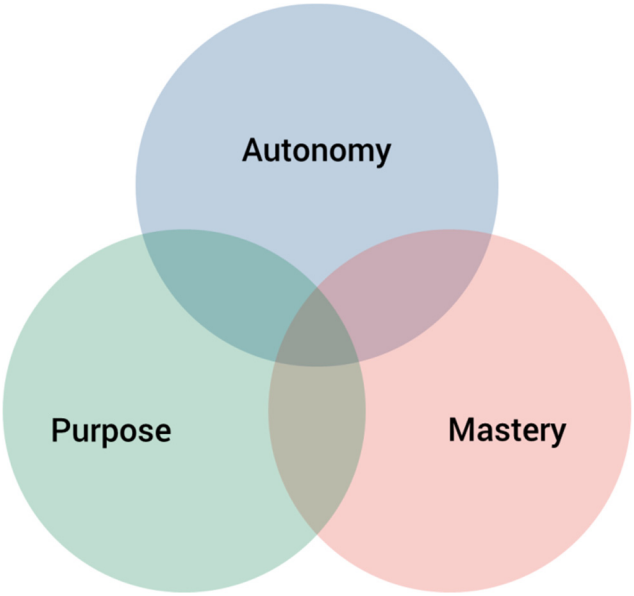

Figure 2: Factors of motivation (Pink 2009)

The agile values and principles support intrinsic motivation much better than traditional phase-oriented approaches. The meaning that employees see in their work depends decisively on the product to be created and is therefore independent of the development paradigm. Autonomy and mastery, on the other hand, rely on the development process.

The principles of the Agile Manifesto explicitly demand autonomy and mastery:

"Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done."

"Continuous attention to technical excellence and good design enhances agility."

"The best architectures, requirements and designs emerge from self-organising teams."

In Scrum, there is no project manager who delegates and monitors activities. Instead, there is a Product Owner who decides independently and on their own responsibility what is to be implemented and in which order, but who does not decide how something is to be implemented. Only the self-organised, autonomous development team decides on the "how".

Pair programming, that is, two developers working together in front of one screen and exchanging ideas continuously during programming, encourages permanent learning among developers. Technical mastery is thus expressly promoted.

1.4. The overall process

The Agile Manifesto describes the values and principles of agile software development. It refers to the development of software, not to its assignment and delivery to the customer. However, agile software development embedded in an organisation in which the assignment or approval process and the delivery of the software take place after cumbersome, slow processes can often only have a very limited benefit from agility: What is the point if new requirements can be introduced into the development process by the Product Owner within a few days and implemented on short notice, but the approval of the necessary funds and delivery of the implemented requirements by IT operations take months?

However, this does not mean that such agility, which is limited to software development, must be useless. If, for example, operational software is to be replaced by software to be developed, the team can obtain feedback from stakeholders in the business departments at an early stage and regularly long before delivery and incorporate it into further development. Only when the new software can do more than the old is the new software delivered. However, the situation is different for software intended for the mass market. There, regular evaluation of the actual usage behaviour and the revenues generated with the software is of crucial importance for further development of the software. An excessively lengthy process for importing new versions stands in the way of efficient feedback of market observations into development and thus effective further development of the software.

According to a Forrester study, most agile development teams were embedded in classic waterfall processes (West 2011):

“Organisations are adopting Agile through a combination of bottom-up adoption and top-down change. But the reality of Agile adoption has diverged from the original ideas described in the Agile Manifesto, with many adoptions resembling what Forrester labels water-Scrum-fall. This model is not necessarily bad, but if application development professionals do not carefully consider and make the right decisions about where the lines fall between water-Scrum and Scrum-fall, they are unlikely to realise Agile’s benefits.”

In order to maximise the full potential of agility, both the assignment or approval and the delivery and thus the cooperation with the IT operation should be agile. The following measures are recommended:

- The contractual agreement between client and contractor allows simple and uncomplicated implementation of new requirements and removal of obsolete ones even during development (see section 1.5).
- The agile principle “Working software is the primary measure of progress” (Beck et al. 2001b) should refer to the delivered, functioning software in front of customers, not to software in the release process. A success is the delivered software (*outcome*), not the milestones reached or the documents created (*output*) on the way there. In contrast to the

classic project manager, the Product Owner in an agile environment focuses more on the business objectives that are to be achieved with the software (see section 1.6).

- The rigid organisational boundary between development and IT operation is softened and overcome. Development and IT operations are together responsible for ensuring that a developed software increment reaches the user quickly and in a quality-assured manner (see section 1.7).

1.5. Assignment of agile software developments

The way in which software is developed, whether agile-incremental or classic-phase oriented, should be reflected in the cooperation between customer and contractor and its contractual arrangement. Fixed-price contracts based on a preliminary specification of all requirements are well suited for phase-oriented developments but are unsuitable for agile development. Service agreements support agile software development much better.

1.5.1. The failure of requirement-based fixed-price projects

In conventional phase-oriented software development, it is generally assumed that the customer specifies their requirements at the beginning in the form of a requirement specification and the contractor prepares a specification sheet in which the technical specifications are defined based on the requirement specification. These two documents form the basis of cooperation and are important contractual components, especially for many fixed-price projects.

However, it turns out that many customers, especially when developing complex products, are not able to identify all their requirements, to describe them precisely enough for a solid cost estimate and to correctly assess the importance of their requirements in advance. An indication of this: according to a study, 45% of the functions were not used at all in four software products and another 19% were hardly used (Johnson 2002).

The idea that the customer specifies a software in advance and then receives the software according to their actual needs at a fixed price often fails for the following reasons:

- Even seemingly simple requirements turn out to be insufficiently specified. For example, the request “A user can enter his phone number into the system” could mean that the user can simply enter something in a field called phone number and that something is stored in the system. However, it may also mean that the system must ensure that it is actually a phone number; that the system must transform the number into a standard format; that the system must check whether the phone number matches the specified location; that the

user must be asked to re-enter their phone number if the first number entered is not valid; that the user may be blocked by the system in the event of repeated incorrect entries; and so on. Depending on the actual requirements, the cost of implementation can vary considerably.

- The customer overlooks requirements. This often happens especially with fundamental requirements that are so obvious to the customer that they no longer notice them.
- Requirements are often unclearly formulated. For example, the requirement "Bring me one litre of milk from the supermarket, and if they have eggs there, bring me six" could lead to six litres of milk – because there were eggs there. Linguistic misunderstandings are a source of misinterpreted requirements.
- During the development period, general conditions such as legal requirements or business processes can change. This may also change the requirements for the assigned software system.
- Technical challenges, especially with regard to quality requirements (performance, security, usability, etc. – see section A in the appendix), often only become apparent in the course of the project. These challenges can be considerable.

Due to these deficits and incompleteness of requirements and technical risks, the estimation of software costs is very uncertain. A cost estimate based on a requirement specification typically

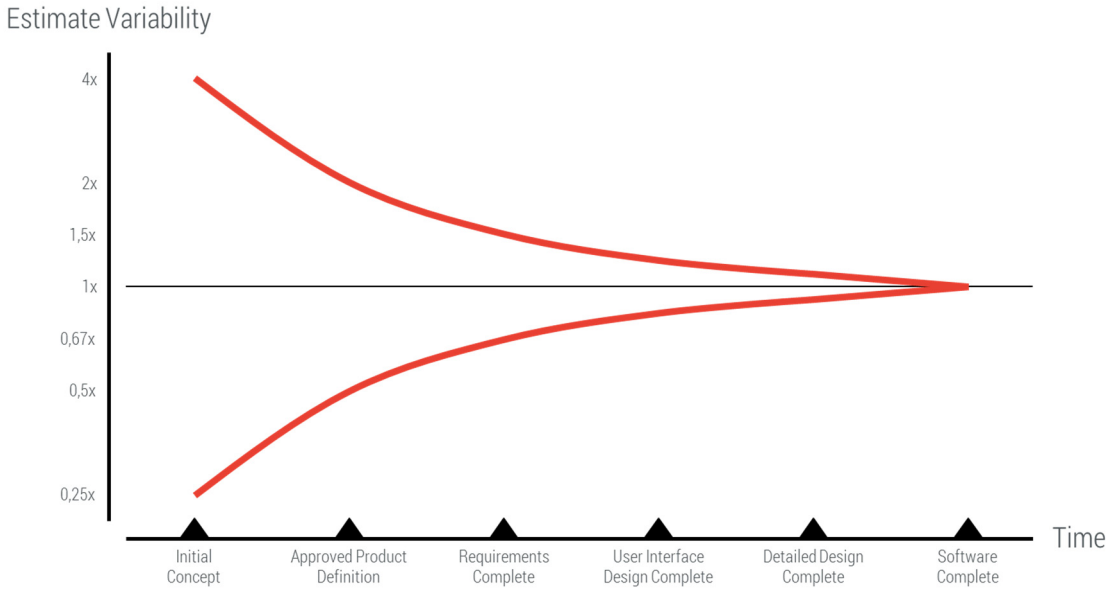

Figure 3: Idealised cone of uncertainty (McConnell 2006)

has an uncertainty of -33% down and 50% up – even with a high-quality requirement specification and an experienced estimator (McConnell 2006). In practice, the uncertainty is often larger.

In conventional project management, the costs are estimated according to the definition of objectives and the specification of requirements. The specification and the estimated costs are the basis

for the project plan. Even if the uncertainty of the estimates is pointed out at the beginning of the project, the initial effort estimate remains the reference and the measure for evaluating the development work. If the efforts were underestimated at the beginning, development managers and the development team are under pressure. This has considerable consequences for the costs and quality of the work.

It is one thing to implement the requirements more or less successfully and get the software up and running, but it is quite another thing to write software of high quality. Good software not only works but can also be developed sustainably and efficiently. It is not enough to design a software architecture and a software design once at the beginning; this must be implemented and questioned and, if necessary, adapted during the entire development phase. High-quality software also includes a consistent programming style, logical naming of classes, methods and variables, and appropriate commenting of the source code. The development team must invest time and resources to ensure software quality.

If the effort involved in phase-oriented development has been underestimated and the customer or the development management puts pressure on the team, the software quality usually suffers. This pressure initially shows the desired effect: The software does what it should. What is not seen are the quality defects “under the bonnet”. And what the customer does not yet know: every further development becomes more and more expensive, and they are trapped in a project that is hardly maintainable and can hardly be passed on to another team.

High-quality software has its price. If there is no willingness to pay this price, especially if price that is too low was promised at the beginning due to underestimated expenses, the software can become much more expensive in the end.

1.5.2. Service agreements

A doctor isn’t paid to cure a patient but to attempt it. This is the characteristic feature of a service agreement: it is not the effort required to implement the scope that counts but the effort required to provide the service. This openness with regard to the services to be provided avoids the need to specify requirements in detail in advance and to calculate the cost of their implementation, and thus also eliminates the uncertainties in cost estimation and the rigidity that lead to problems with fixed-price projects.

However, from the point of view of the client, service agreements seem to be a bottomless pit. They commission the contractor without knowing what it will ultimately cost to achieve the goals. They know that it will be in the interest of the contractor to provide as many billable services as possible – whether useful for their goals or not – and that they can never be sure that services have actually been provided for the hours billed.

But the loss of control of the client in service agreements in comparison to requirement-based fixed-price agreements does not exist at all: the client has only very limited real control in complex fixed-price projects as well.

In the case of service contracts, there is no need for the client to provide detailed requirements for the award of the contract at the start of the project. This does not mean that requirements engineering is not necessary for service contracts, but it does mean that requirements engineering is oriented to what is necessary from a business point of view and not to what is necessary for drafting contracts. It also allows much more flexible handling of change requests during the project term, since contractual changes, even in the form of change requests, are not necessary. Requirements engineering can thus be made much leaner and more effective from a business perspective. However, a precise description of the project goals should not be dispensed with, as these goal definitions serve as orientation and guard rails during the project precisely because of the rudimentarily recorded requirements at the beginning of the project.

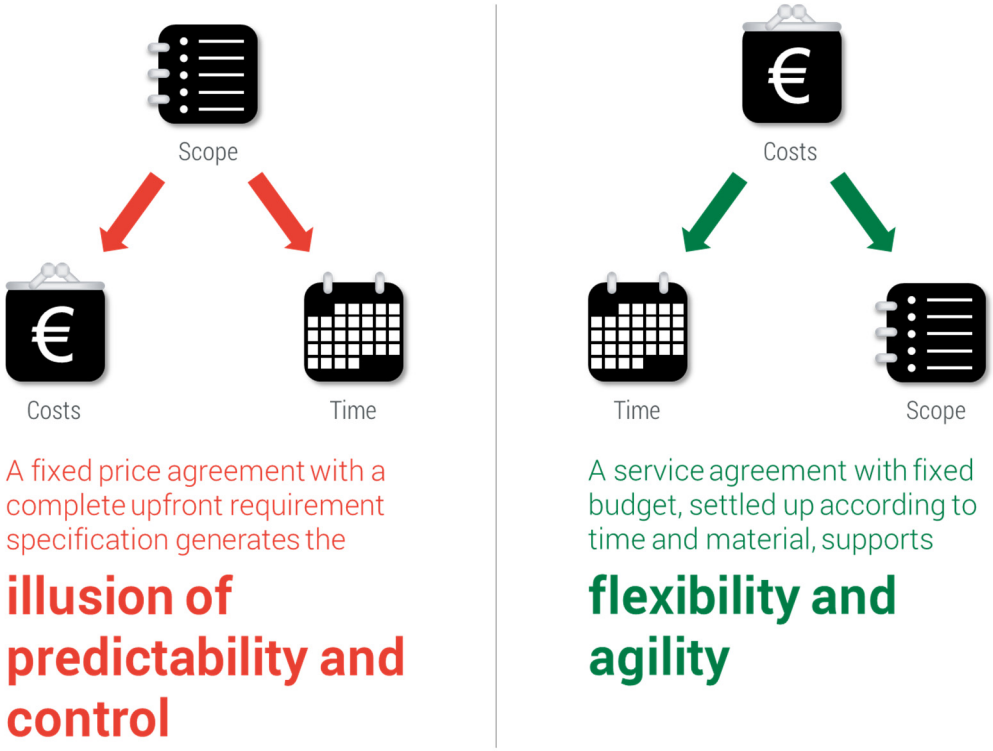

Figure 4: Rotation of the triple constraint

1.5.3. Agile fixed price

With conventional phase-oriented software development, the time and costs for implementation are derived from the fixed requirements. The aim is to determine the costs and completion date as accurately as possible, which, however, cannot usually be reliably predicted in practice due to the great estimation uncertainty.

With agile service agreements, on the other hand, the budget and time can be fixed in advance, and the scope of services is increasingly specified during the development period. In this case one speaks of an agile fixed price.

1.5.4. Agile contract for work and labour

Internal company guidelines may require that software contracts, whether to internal or external contractors, may only be awarded in the form of contracts for work and labour. The characteristic feature of a contract for work and labour is the acceptance of the assigned labour by the client. A client can only accept what has been specified beforehand. That means that any form of contract for work and labour would seem to prevent agile implementation.

One way of enabling agile software development even in the context of contracts for work and labour is to define the requirements for short, successive development iterations, so-called *sprints*. The work contract itself does not contain either the individual requirements or the acceptance criteria but merely describes the acceptance process and defines the binding acceptance after each sprint. This fulfils the requirements of a contract for work and labour.

1.5.5. Excursus: Provision of temporary workers

Software is often developed by or with the support of external partners. In this context, the topic of temporary employment must be considered. If there is no formal hiring out of employees, the contracting company must ensure that requirements are only introduced into the development process via the contractually agreed legally compliant communication channels. This must be checked on a case-by-case basis and depends on the constitution of the team, the agile framework and the contractual relationship.

For example, if Scrum is used and the development team is completely provided by an external service provider, the client and the contractor could agree that a representative of the service provider who is allowed to take orders according to the contractual agreement is present and confirms the scope of the Sprint Planning I, that is, when determining what the development team is to implement in the upcoming sprint. Formally, it is not the development team and the Product Owner who agree on the scope of the sprint but the designated representative of the service provider and the Product Owner – with expert advice from the development team. During a sprint, the Product Owner may not submit any new requirements, at least not without the involvement of the service provider representative. However, this complies with the Scrum rules anyway. The Product Owner must be available to the development team for questions. If internal company directives do not allow this either, agile work is not possible.

If a development team consists of both external and internal employees, agile development should be impossible, as tasks can no longer be jointly accepted and answered for by the development team.

1.6. The role of the Product Owner

The conventional project manager has the task of implementing predefined requirements while adhering to the given time and cost constraints. Their focus is primarily on internal costs and efficient and timely achievement of milestones.

The Product Owner is different, as provided by some agile frameworks such as Scrum. They focus less internally than externally on the expected benefits of the product increments to be delivered. They are not obliged to implement software according to specifications in accordance with the budget and milestone plan but decide independently and on their own responsibility when which requirements are to be incorporated into development. The aim here is to achieve the overarching economic goals with the means available.

This independent decision-making competence attributed to them can lead to problems, especially in traditional organisations in which the stakeholders are used to demanding their requirements at the beginning of development. Stakeholders often exert pressure on the Product Owner on the way to escalation and demand the implementation of their "own" requirements. This pressure often encourages Product Owners to promise individual stakeholders the scope and timing of their requirements. However, this undermines the agile approach: stakeholders expect the promises made to be kept, which requires more comprehensive planning and considerably limits the Product Owner's decision-making authority.

It is better if the Product Owner clarifies the various requirements and coordinates them with the stakeholders. The Product Owner thus takes on the role of a moderator who mediates between the various interests. A number of methods, such as Planning Poker[2], are available to the Product Owner for this purpose. The aim is always to determine the expected added value together with the stakeholders by implementing the different requirements and thus determine an economically reasonable order of implementation. However, the disadvantage here is that frequent and possibly very time-consuming coordination of stakeholder interests is necessary to ensure sufficient agility. In addition, the quality of the software and the ability to innovate often suffers: the requirements identified and coordinated in this way are mostly functional and rarely non-functional in nature; the group tends to agree on the lowest common denominator and to avoid innovations.

Ideally, a Product Owner should only assume the moderating role in requirements gathering. The decision as to which requirements are to be introduced into the development process and when should be solely their responsibility. However, this does not mean that they are not generally accountable to anyone: A Product Owner should be measured against the overall economic goals

[2] Planning Poker is a method of estimation in a team of experts. Each team member has a set of cards with possible estimates. First of all, each member estimates for himself or herself and discards the corresponding card face down. Then everyone reveals their cards and thus their estimates. In the case of different estimates, the two members whose values are the furthest apart discuss. Then a new estimation round takes place. This is repeated until there is a consensus. Planning Poker is often used to estimate costs in an agile environment. It can also be used to estimate the value of requirements.

they have to achieve with the means made available to them and within the system context of the product for which they are responsible. Determining meaningful key performance indicators that best reflect the economic success of the product is the best basis for the effective work of the Product Owner. This is the best way to ensure that the Product Owner is geared toward the economic *outcome*, not to the *output* of any intermediate results that are not directly relevant to the user.

1.7. DevOps & QM

Even if the Agile Manifesto explicitly refers to software development, agility can only really come about if the Product Owner can not only develop the developed increments promptly but also have them delivered promptly. This requires that the software components can be efficiently integrated into an overall system and that the quality of the software is tested largely automatically before delivery. The classic organisational separation of development, quality management and IT operations with their defined handover points no longer meets these requirements in an agile environment. Due to the highly dynamic nature of agile development, development, quality management, IT operations and all other organisational units involved in a rollout must grow closer together. *DevOps* arose from this line of thinking.

The word *DevOps* is composed of the words *Development* and *Operation* and is, according to Bass et al (2015):

"[…] a set of practices intended to reduce the time between committing a change to a system and the change being placed into normal production, while ensuring high quality."

DevOps deals with the organisational integration of the IT units *development*, *quality management* and *operations* and the tools and processes that support efficient and quality-assured integration (*continuous integration*) and delivery (*continuous delivery*) of the software. Similar to the term *agility*, the term *DevOps* is used inconsistently. In particular, the distinction from *continuous delivery* is blurred. Continuous delivery tends to focus primarily on the process and the supporting tools required for efficient delivery, while DevOps also includes the organisational aspect. Together, however, they have the goal of quickly providing the end user with a quality-tested product – as formulated by Bass et al.

This goal can only be achieved if there is a continuous, highly automated process, starting with automated tests of the software units and components, through automated integration of the system and regression tests, to automated software delivery including final tests in the production environment with subsequent activation. This process cannot only be started when the software is taken over by IT operations, nor can IT operations or quality management be solely responsible

for this process in parallel and independently of development. Rather, development, quality management, IT operations and all other departments involved must work hand in hand.

While the conventional IT organisation has a defined transition point at which IT operations takes over the software created by development and then delivers it, IT operations now provides the entire range of tools that enables delivery to be as automated as possible. The notion of a "finished" product that only needs to be maintained by IT operations after testing by quality assurance and subsequent delivery makes no sense in an agile environment. Rather, development, maintenance and operation take place in parallel and in close organisational coordination. The yardsticks for the organisational division of tasks are not the functional units "development", "quality assurance" and "IT operations" but at best product modules for which interdisciplinary teams are responsible. This poses major challenges for many classic companies, especially in the automotive industry.

The new role of quality management by DevOps and the testing of software in an agile environment will be discussed extensively in Part 2.

1.8. Agility beyond software development

The title of the Agile Manifesto is: "Manifesto for Agile Software Development". It refers explicitly to the development of software. This is different with Scrum. The Scrum Guide states: "Scrum has been used to develop software, hardware, embedded software, networks of interacting function, autonomous vehicles, schools, government, marketing, managing the operation of organisations and almost everything we use in our daily lives, as individuals and societies." (Schwaber, Sutherland 2017).

For what purposes does an agile approach make sense? What are the prerequisites for it?

1.8.1. Incremental-iterative vs. phase-oriented

Agility makes sense when the target is vague and the task is complex. An incremental-iterative development is then the best approach, as the initially unclear goal is continuously sharpened and most effectively achieved. However, one condition must be met: The effort for changes to what has already been created must be suitable. If, for example, the client wants the user to be able to configure the colours of the software in a late development phase, this requirement that was not originally planned may require some modifications. The modification costs are the price for an incremental approach and must be economically justifiable in comparison to the value. This is the case for most software requirements.

On the other hand, the modification costs for many hardware developments are much higher and therefore do not always justify an incremental approach. For example, production tools are often expensive and would have to be adapted to changes, and the B-pillar should not always be repositioned during product development in vehicle construction. Nevertheless, there are also non-software products that can be developed incrementally, such as the introduction of 3D printers to enable cost-effective adaptation of components, thus enabling agility. There are also software developments that are better phase-oriented. There are software developments with both clear objectives (e.g., the 1:1 transfer of an existing iPhone app to Android devices) and high modification costs (e.g., changes to basic Java interfaces used by the entire Java community). In this case, a phase-oriented approach may be appropriate.

The following criteria generally apply:

The incremental-iterative approach should be taken into consideration when the target is blurred (see Stacey 2000 and others; Figure 5). If the goals are clear and a team can achieve them through clearly described processes, agility is not useful. The production of bulk goods and routine processing are therefore rather unsuitable for an agile approach.

It is always necessary to weigh the following: Is it really possible to determine the goal and the achievement of the goal precisely and economically enough in advance? Do the costs for subsequent changes necessitate detailed planning at the beginning? With many product developments, this cannot be answered clearly with yes or no. Here it is necessary to consider and, if necessary, to implement some partial developments in a phase-oriented manner and others in an agile manner.

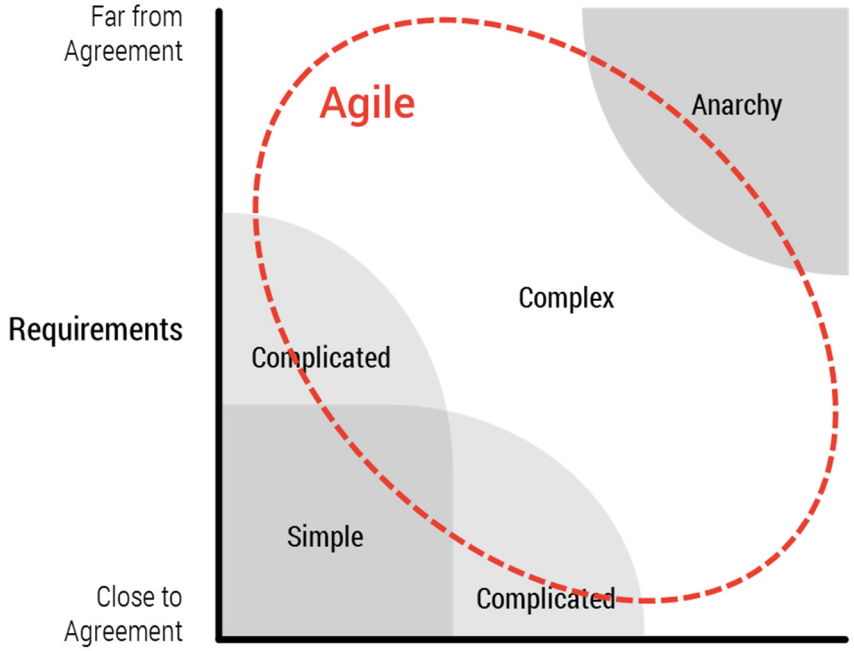

Figure 5: Stacey Matrix

1.8.2. Self-organised vs. instruction-oriented

Self-organisation means that the development team itself decides how to implement the requirements and the client's representative (the Product Owner in Scrum terminology) as part of the team decides which requirements are implemented in accordance with the overall business objectives. Self-organisation enables the team to make quick and uncomplicated decisions when implementing a complex task incrementally and iteratively. Self-organisation is a prerequisite for agility. Self-organisation requires that the team members want to be responsible, that they have the qualifications to take responsibility and that they are given the freedom to do so. On the other hand, agility is hindered or even prevented if one of the following situations occurs:

- A real know-how gap: If the knowledge necessary for the development lies exclusively with the superior or project manager and the team members do not have the qualifications to carry out the development independently, self-organisation cannot succeed. An agile team (possibly temporarily expanded to include supporting experts) must have the expertise to be able to take responsibility for the quality of the product.
- A perceived know-how gap: The knowledge gap between the manager (superior or Product Owner) and the other team members is often only perceived by the manager and is not real. However, this is also a real obstacle to self-organisation. A manager who considers themselves indispensable and engages in micro-management will not allow their employees to work on their own responsibility.
- Avoidance of responsibility: Self-organisation means taking over responsibility by the team within the scope of the set objective. For this the team must be ready and accept that the convenient means of delegation of responsibility upwards ("escalation") is often no longer available. Likewise, everyone must accept that the results of the entire team are evaluated, not the individual contribution of the individual. If the team is not prepared to do this, self-organisation will not work.
- Restrictive conditions: Legal or organisational restrictions can severely hinder and possibly prevent self-organisation. In the automotive industry, for example, there is an increasing tendency due to adjudications to prescribe the cyclomatic complexity of software as a binding quality criterion. It should not be denied here that cyclomatic complexity can be an indicator for the maintainability of software, but neither low cyclomatic complexity guarantees good maintainability nor does a high cyclomatic complexity need to be bad in particular cases. Such provisions have the effect that the team obeys rules rather than taking responsibility for quality itself.
- Interaction of many disciplines: Agility is a tried and tested approach, especially for complex developments that are difficult to plan. However, agile development becomes difficult if many different teams of different disciplines are to contribute to the overall product. Self-organisation is only practicable up to a certain team size (typically up to nine members). Division of the teams is also common practice in agile software development

and is successfully applied (see section 1.9). However, if some of the teams cannot or do not want to work in an agile manner for the reasons mentioned above, it must be investigated in individual cases whether agility can still function and makes sense for the overall development. The possibility of working in the individual teams on an agile basis should not be impaired by this.

1.9. Scaled agility

Agility can only be lived by self-organised, cross-functional teams whose members are committed to the values of self-commitment, courage, focus, openness and respect. Internal hierarchies stand in opposition to this, so agile teams can therefore not be scaled arbitrarily. According to Scrum, the development team may not exceed nine members.

However, large development projects must be handled by more than one team. At present, various models for organising multiple agile teams are being discussed, some of them quite contensiously.

The foundation of agility scaling is agile teams. If several agile teams are working on a product, the teams should be put together in such a way that their independence is preserved as far as possible. The coordination effort between the teams should be minimised. Typically, this is best achieved by building feature teams responsible for functional units. Teams that specialise in individual technologies (e.g., GUI or database development), however, are unusual in an agile environment and usually not effective.

In extreme cases, it may turn out that the feature teams do not have to coordinate with each other at all. In this case, there is actually not one product but several products, each of which is developed autonomously by a team. Coordination of the teams is therefore not necessary and would be a waste.

However, if decoupling agile teams is not possible, two scaling patterns are available (see also Foegen and Kaczmarek 2016). The following patterns are based on Scrum teams as agile teams. It is possible in principle to transfer them to other agile frameworks.

1.9.1. Tightly coupled teams

This pattern has a Product Owner and a product backlog, whose items are implemented by several development teams. The teams' sprints are synchronised and begin with a joint Sprint Planning I of all development teams, in which the product backlog items are assigned to the development teams. Each development team then carries out its own Sprint Planning II. Representatives of the development teams coordinate their activities daily during a sprint in so-called *Scrum of Scrum*

meetings. The development teams present their results to the Product Owner during the sprint review. A sprint is concluded by team-internal retrospectives followed by a joint retrospective with all development teams.

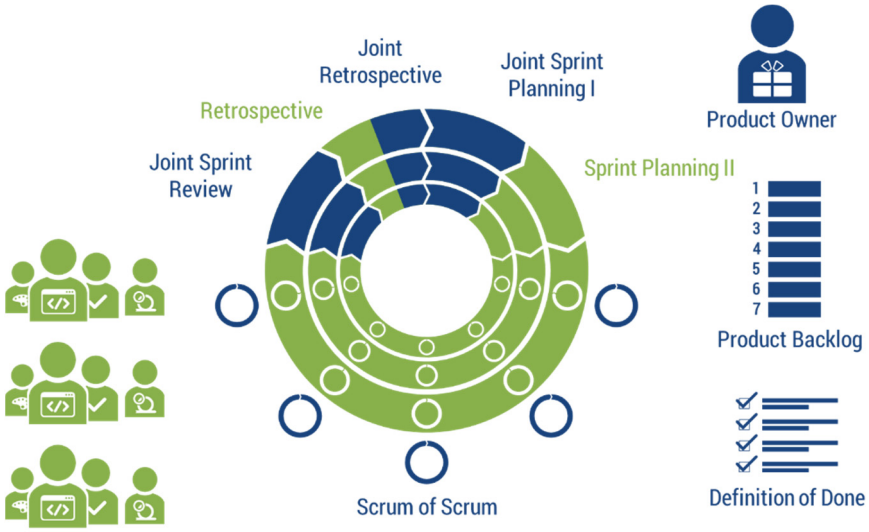

Figure 6: Pattern of tightly coupled agile teams

This pattern assumes that a single Product Owner is able to provide several development teams with sufficient product backlog items. Since a Product Owner can delegate work to the development team, this should be possible for up to four development teams. The *Large Scale Scrum* (LeSS) framework, which is based on this pattern, even assumes that up to eight development teams can work together on a product in this way (Larman and Vodde 2018).

1.9.2. Loosely coupled teams

In contrast to tightly coupled teams, each team has its own Product Owner and determines when which events such as sprint planning or retrospective take place. Teams are therefore more autonomous; team representatives only coordinate during the sprints in the form of Scrum of Scrum; the results of all teams are presented and discussed in joint sprint reviews. There is an additional level for this: Every 1 to 3 months, the teams plan and evaluate their work together with a Chief Product Owner and a Unit Agility Master. This means that there is a hierarchy for both the Product Owners and the Scrum Masters. The Chief Product Owner is responsible at the top level for the product backlog, which is then broken down into staged backlogs for the different team Product Owners during stage planning.

The hierarchisation of requirements management provides planning security and thus restricts the autonomy of the individual agile teams. Therefore, it must be seriously questioned whether such an organisation is still agile itself. The well-known *Scales Agile Framework* (SAFe), which is

based on this pattern, is criticised for not complying with agile principles and values (Schwaber 2013).

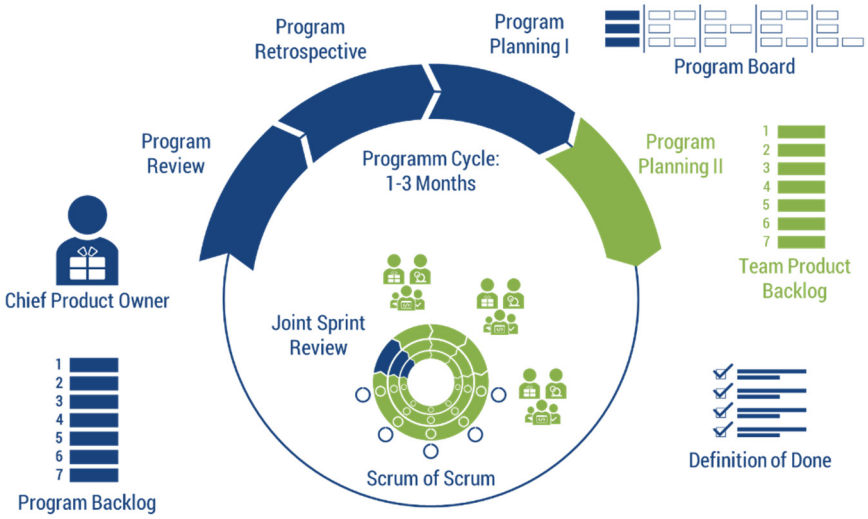

Figure 7: Pattern for loosely coupled agile teams

1.10. Conclusion

Agility is a useful approach for development teams to develop software effectively. In recent years, many companies have successfully developed software in an agile manner, saving costs and improving the quality of their products compared to the conventional, phase-oriented approach. Agility is indispensable in many IT companies today.

The benefits of agile software development are only fully realised when the entire process, from the assignment of the development to the delivery of the software, supports agility. This means that a cross-functional team specifies requirements iteratively, develops software incrementally, delivers increments continuously and optimises agile practices iteratively.

Agility can also be appropriate beyond software development, especially in complex product development and other challenging activities. However, the prerequisite for this is that the benefit of agility justifies the costs for the conversion of existing increments economically.

The development of extensive products requires the coordination of several agile teams. Currently there are different approaches for scaling agile developments. So far, no framework has been able to establish itself as a standard. It is unlikely that it will be possible to implement a generally applicable scaling model as best practice in future due to conditions that differ. The extent to which scaling models support agile values and principles is currently under discussion.

2. AGILITY AND SOFTWARE QUALITY

The aim of software development is the implementation of software requirements. These also include quality requirements in addition to functional requirements and boundary conditions. This applies to agile as well as conventionally developed software. The consequences on software quality of switching to an agile approach and how quality can be achieved in an agile environment are explained below. It turns out that agility is a paradigm shift not only for the actual development but also for quality management.

Agile methods are now also used beyond software development. However, this part is to explicitly focus on quality management in agile software development, since the general applicability of the agile approach in the development of software is undisputed and agile software development is mature and widespread to the greatest extent.

Software quality includes a number of quality characteristics. ISO/IEC 25010 describes a system for quality characteristics of software products (ISO/IEC 2010). The 8 quality characteristics and a total of 31 sub-characteristics are listed in the appendix. In the following, they serve as a reference to illustrate the relationship between the quality measures and software quality.

2.1. Quality through agility

The Agile Manifesto says: “Working software over comprehensive documentation.” This suggests that an agile development team is primarily concerned with the implementation of the required functions; quality characteristics, e.g., the maintainability of the software by a proper system documentation, seem to be of less importance. Self-organisation and thus the lack of binding specifications by management regarding the quality standards to be met also reinforce the impression that no high quality of the software is to be expected. Neither the Agile Manifesto nor frameworks such as Scrum have explicit QM roles, nor do they require the principle of dual control to ensure quality. Rather, the quality of the software essentially depends on the quality of the agile team: the Product Owner, who introduces requirements into development – also with regard to quality and legal boundary conditions – and the development team, which is independently responsible for their implementation.

For stakeholders outside the agile team, this means a loss of control. Although quality requirements are also specified and validated in an agile setting (see section 2.5), the development team is responsible for the quality of implementation. The omission of quality management which intervenes from the outside in regard to how the agile team develops software does not have to result in a deterioration of the software quality under any circumstances. The introduction of agility alone can have a positive effect on quality, as explained below.

2.1.1. Motivation in an agile team

Agility encourages autonomy and mastery of the agile development team and thus motivation of the team and each individual team member (see section 1.3). A motivated development team has the ambition to develop products of high quality. The quality of the implementation is not demanded from outside but desired by the team itself. This is an important prerequisite for developing high-quality software – regardless of its quality characteristics.

2.1.2. Early user feedback

Through incremental development, user feedback can be captured early and repeatedly and incorporated into further development. Unnecessary and thus quality-reducing requirements are implemented less frequently. The quality from the user's point of view is continuously improved. Phase-oriented development implements the requirements that were considered relevant at the beginning. In contrast, agile development implements the requirements that are actually relevant. The quality characteristics *functional completeness* and *functional appropriateness* are therefore usually realised much better with an agile approach than with the conventional phase-oriented approach.

2.1.3. Adequate time for implementation of quality requirements

Classic project management assumes that the duration and costs of software development can be determined based on a complete specification. However, if the actual effort is greater than expected, the development team is put under pressure. This pressure then often comes at the expense of quality.

The *maintainability* of software is affected in particular, as this can hardly be verified for the client. All other quality characteristics that cannot be directly verified from outside, such as *security* or *portability*, can also suffer from deadline pressure.

There is no complete specification in advance for agile software development. The implementation velocity becomes apparent in the course of the first iterations. The development team can constantly keep software quality high, and the resources for the development or functional scope of the software are adjusted as required.

2.1.4. Continuous integration, continuous delivery

In classic software development projects, the previously developed software modules are often integrated into a complete solution late and just before the planned delivery, and the software is delivered much less frequently than in agile development. This often leads to considerable difficulties during integration: The modules that were previously created for months do not fit together as expected and must be re-engineered under great time pressure. Agile development teams, on the other hand, integrate software much more frequently. Continuous refactoring, integration and delivery continuously improves the *maintainability* and *installability* of the software.

2.2. Principles of agile quality management

Agile software development, like conventional software development, requires management of the quality of the product. This agile quality management follows the same agile values and principles as the agile software development itself. The way of managing software quality thus differs fundamentally in an agile environment from the quality assurance of phase-oriented development, which often only checks the quality of the product after development as an independent organisational unit.

By contrast, the following principles apply to agile quality management:

2.2.1. The agile team is responsible for quality

An agile principle is:

"The best architectures, requirements, and designs emerge from self-organizing teams."

(Beck et al. 2001b)

With Scrum, the following applies:

"Scrum Teams are self-organizing and cross-functional. Self-organizing teams choose how best to accomplish their work, rather than being directed by others outside the team. Cross-functional teams have all competencies needed to accomplish the work without depending on others not part of the team."

(Schwaber, Sutherland 2017)

The agile team is responsible for the developed product, including quality. Within the team, however, a distinction must be made between domain-oriented quality requirements and technical implementation quality: It is the responsibility of the Product Owner to introduce the quality

requirements necessary and meaningful to the business into development and the responsibility of the development team to implement requirements with the required quality. For example, the Product Owner could formulate data protection requirements. However, how these domain-oriented requirements are technically implemented is the sole responsibility of the development team. It would also be conceivable, for example, that a functional safety department would demand safety requirements for the product from the Product Owner. The following also applies in this case: before delivery, the Product Owner or the functional safety department checks the implementation of the domain-oriented requirements, not the kind of technical implementation.

Therefore, a distinction must be made between testing within the development team – if necessary with external consulting support – and quality management outside the team, which sets and validates quality requirements on a purely domain-oriented and business level.

This principle is crucial for agility:

- If responsibility for implementation quality or certain aspects of implementation quality were outsourced from the team, the organisational separation of development and quality management would slow down the overall process.
- Outsourced management of implementation quality would generally be responsible for several agile teams. The quality guidelines would usually apply to all teams and the special circumstances in the individual teams would not be sufficiently taken into account. Only the development team, which works on development on a daily basis, knows all the details in order to be able to adequately assess the particularities.
- An external quality management system which is only responsible for certain quality characteristics will tend to overemphasise these characteristics. There is a danger of calling for perfectionism at the very beginning of development, which stands in opposition to the idea of the shortest possible time-to-market.

2.2.2. Quality arises during development

In phase-oriented software development, development and testing of software are regarded as two separate activities, such as in the V-model, which is a common procedure model in conventional software development. For example, in the V-model-based Automotive SPICE, *SWE.3: Software Detailed Design and Unit Construction* and *SWE.4: Software Unit Verification* are defined as two separate processes (A-SPICE 2017).

In practice, a phase-oriented approach usually means that software at component level is initially tested less but more intensively the closer the delivery gets. Quality assurance in the classical context focuses on verification of the completed software (based on the specification) and release for rollout.

However, as early as 1986, W. Edwards Deming, for example, called for quality not to be considered afterwards:

"Inspection does not improve the quality, nor guarantee quality. Inspection is too late. The quality, good or bad, is already in the product. As Harold F. Dodge said, 'You can not inspect quality into a product.'"

(Deming 1986, p. 29)

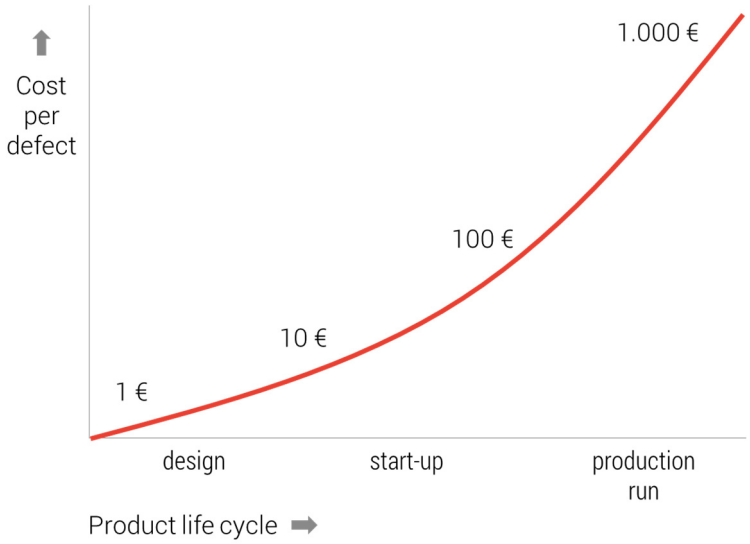

Figure 8: Rule of Ten – At each step in the value chain, an error can be corrected about ten times faster and cheaper than in the next step

In agile software development, quality is considered as early as possible and – as far as possible – tested during development in order to keep the effort for correcting errors as low as possible (see *Rule of Ten*, Figure 8). The Product Owner defines acceptance criteria (if necessary with the support of the development team or stakeholders) before an iteration. These criteria serve as a benchmark for successful implementation. During an iteration, developers typically program unit tests before component development, which are then used for automated testing. This *Test-Driven Development* (TDD) helps to ensure that requirements are sufficiently clarified between Product Owner and development team before implementation and that the developers focus on the implementation of the requirements defined in the unit tests. Since the unit tests are written in the same programming language as the software to be developed and are usually integrated into the development environment, test programming, software programming and automated testing go hand in hand. The developer performs these steps autonomously in cycles of only a few minutes in some cases. Software development and testing on the unit level merge into a single step.

Unit tests are the centerpiece and foundation of agile quality management, especially of functional verification. Further tests, such as functional tests of the integrated system or GUI tests, also take place in an agile environment, but usually to a lesser extent than in conventional development. Unit tests are easy to implement, robust and perform well. The closer to the business and less technical a test method is and thus the more closely it corresponds to real conditions of use, the more unstable and slower the tests are and the more complex their implementation.

The test scopes should be built on the respective integration layers like a pyramid (see Figure 9). However, the type of tests to be focused on also depends strongly on the type of product and the requirements. In addition, the tests should be coordinated with each other on the different integration levels. For example, tests on the unit level should not simply be repeated on the UI level, but exploratory tests should be carried out on the UI level under different conditions of use that are not possible on the unit level.

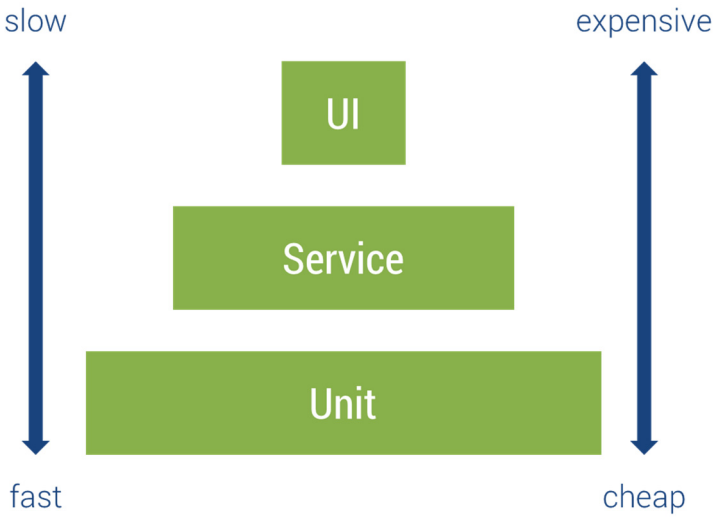

Figure 9: Test pyramid with the fast and inexpensive unit tests as a foundation

Various test procedures and their relevance in an agile environment are described in more detail in section 2.3.

2.2.3. No micromanagement

The Product Owner is part of the agile team, but not part of the development team. The development team is autonomous, and the Product Owner is not a head of development who tells the development team how to develop the software. Neither the Product Owner nor anyone outside the agile team controls the development team through micromanagement.

The Product Owner identifies and prioritises requirements according to business goals. They focus on the validation of the software that is deliverable and/or delivered. Work products that cannot be delivered, such as specifications or concepts, are neither standards for quality nor objects of quality management. In contrast, the Product Owner continuously measures and evaluates the business success of the delivered product and derives functional and non-functional requirements from this.

Section 2.5 describes in more detail how quality can be demanded and tested by the Product Owner and other stakeholders without restricting the autonomy of the development team.

2.2.4. Summary: Tasks of quality management

Quality management in an agile environment exists in two different forms: one within the agile, autonomous development team, which verifies the implementation of the requirements, and one outside the development team, which has an outward view of the use of the product by the users and validates the domain-oriented quality.

Quality management in the development team is dealt with in detail in the following section.

Furthermore, it can make sense to involve specialists in the work of the development team temporarily or in an advisory manner regarding certain quality characteristics. These quality expert pools are described in more detail in section 2.4.

Section 2.5 describes how Product Owners and stakeholders set quality requirements and test the quality of the software without undermining the autonomy of the development team.

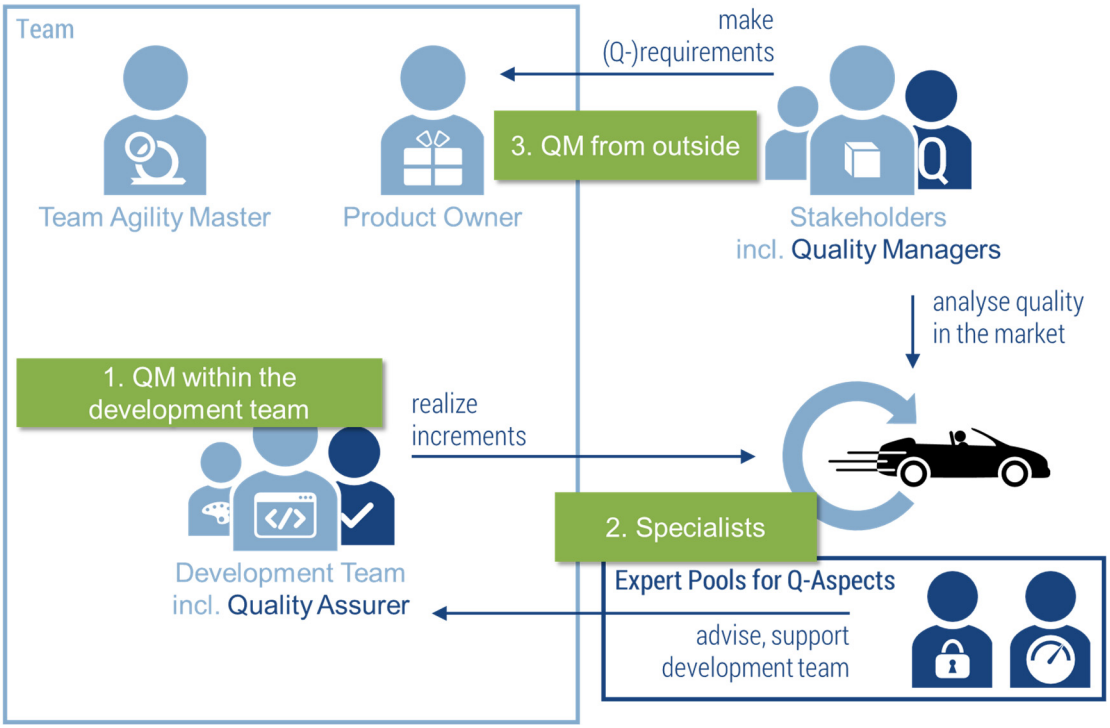

Figure 10: Overview of quality management tasks in an agile environment

2.3. Agile testing in the development team

2.3.1. Implementing and testing requirements

The implementation of a requirement by an agile team begins with the Product Owner describing the requirement – in the Scrum context the so-called *Product Backlog Item* (PBI). Typically, the requirement is formulated as a *User Story* in the form "As a <type of user>, I want <some goal> so that <some reason>".

Before a Product Owner steers a PBI into development, they should define acceptance criteria, if necessary with the support of the development team or stakeholders. These criteria serve to clarify the requirement and help with later acceptance. Acceptance criteria are functional test cases in most cases but can also affect quality properties such as time behavior or usability characteristics. The primary purpose of the acceptance criteria is to clarify the requirement more precisely. User story, acceptance criteria and the relevant discussion with the stakeholders and in the team replace detailed specification of requirements as envisaged in conventional software development.

The actual implementation of the requirement is carried out by the development team. For the purpose of Test-Driven Development (TDD), before implementation the development team can define functional test cases, such as ones based on acceptance criteria, and include them in the test tools for test automation. In addition, prototypes can be easily created with a pen and paper, for example. Prototypes also primarily serve to clarify the order.

A unit is implemented by a programmer. The programmer implements the test cases for this unit as unit tests, puts the first functions of the unit into effect, tests the unit, brings about further functions, re-tests, etc. This is done in short iterations, often lasting just minutes. To test the unit, the programmer only has to trigger the unit tests implemented in the same development environment as the unit, which then automatically test the unit. Unit tests are the smallest test unit. They are relatively inexpensive, robust and quick to implement.

Pair programming originates from the agile framework *Extreme Programming* (Wells 1999) and sometimes is also used in other agile frameworks. In pair programming, two members of the development team – usually two programmers – sit in front of one computer and develop a unit together. This is to promote the exchange of knowledge between the team members and improve the quality of the software through joint inspection. Instead of a second programmer, a quality assurer can also assist in finding and formulating unit test cases.

In addition to the unit tests, tools for static code analysis are available to the developer. These analysis tools automatically examine the source code for possible programming errors such as memory leaks or performance issues.

The programmer checks the finished module into the repository, in which the entire source code is versioned and managed. This allows the module to be integrated into the overall solution and the integration tests to be started. The integration tests usually consist of function tests without

integration of the user interface, so that they run efficiently and robustly. In an agile environment, integration does not take place just before a release date, as in phase-oriented development, but much more frequently, usually at least once a day.

The integrated software can then be tested further in various ways. The following list contains typical test methods without any claim to be exhaustive (see a detailed description in Crispin/Gregory 2009):

Explorative testing: A quality assurer tries to provoke unwanted software behavior by using or abusing the software in an unusual way. Exploratory testing is always done manually, even if tools can support such testing.

GUI tests: GUI tests are functional tests in which the graphical user interface (GUI) is also tested. Such tests can be automated. However, such tests are slow, time-consuming and error-prone. Therefore, they should only supplement unit tests and functional tests without a GUI check.

Performance and load tests: Performance and load tests are used to test the time behavior and the behavior of the system under heavy load. Such tests should take place in an environment that largely corresponds to the production environment. The tests are usually tool-based and automated. The development of such tests requires special knowledge.

Regression tests: If tests are repeated regularly, they are called regression tests. In most cases, regression tests are automated. For performance reasons, not all automated tests may be regression tests.

Security tests: Similar to performance and load tests, special expertise is also required for security tests. The tests are both tool-based and exploratory.

Usability tests / persona tests: Usability and persona tests are used to test usability and functionality for completeness and appropriateness. The tests are performed manually.

The development team decides which of these tests to perform and to what extent. If the software has passed these tests, further tests can be carried out even in the production environment after installation before the software is activated (so-called blue-green deployment)[3].

Finally, the Product Owner decides whether they accept the implementation of the requirements. The yardstick here is the acceptance criteria they previously formulated. The Product Owner decides whether the software increment produced is delivered.

The delivered software can be used to test hypotheses (so-called *A/B tests*). Such tests are used in particular to test *functional suitability* and *usability*. For example, an alternative operating concept is implemented and delivered, but only some of the users are activated for it. By analysing the

[3] In a blue-green deployment, the existing software continues to be operated and used, for example in the blue environment, while the new software is installed and tested in a second environment – in this example the green environment. Once all tests have been successfully passed, the requests to the system are directed to the green environment. The blue environment is inactive after all remaining activities have been completed. When a new update is imported, the blue environment is used for installation and testing, etc.

usage behaviour, it can be decided whether the business goals are better achieved with the new operating concept than with the previous one. A/B tests are based on observing usage behavior and are therefore part of test methods described in more detail in section 2.5.

2.3.2. Test automation and the staging pipeline

Test automation, continuous integration and continuous delivery are typical in an agile environment but can also be used in conventional phase-oriented software development. Due to the high level of integration and delivery frequency in agile development, many of the methods and tools for quality-assured, automated integration and delivery of software were developed in an agile environment and first used there.

The process described above for testing the software can be extensively automated. These include unit tests, static code analysis, function tests (with or without GUI tests), performance and load tests, blue-green deployment and A/B tests. In contrast, explorative, usability and persona tests cannot be automated. Even if tools support quality assurers, it is still the task of quality assurers to perform them manually.

The automation of security tests is also difficult in some cases. Security tests consist of static code analyses that can be automated and manual exploratory penetration tests by security specialists. Here, too, tools are available that support but cannot replace manual work.

The development team uses the different test methods, whether automated or manual, usually at different times during the implementation of a requirement. Accordingly, the test methods are used in environments that differ.

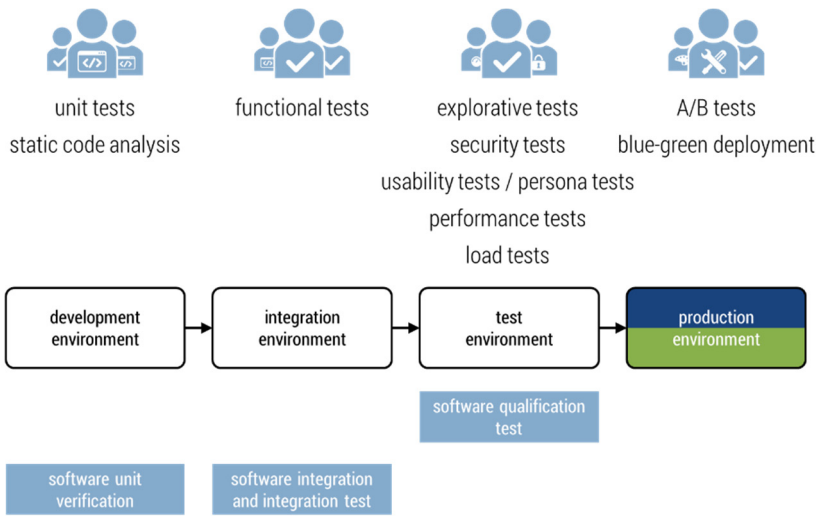

Figure 11: Example of a staging pipeline. The typical test methods are listed for each environment.

Figure 11 shows possible system environments. The actual software development, the development and execution of the unit tests and, if appropriate, static code analyses take place in the development environment, which is typically operated locally by every developer on their computer. The integration environment is primarily used by quality assurers to test functional integration. Further tests are carried out in one or more separate test environments. This primarily involves quality assurers, who may be supported by quality specialists (see section 2.4). Blue-green deployment and A/B tests may take place in the production environment.

The environments, the test methods described here and their allocation to each other are exemplary. Depending on the product, organisation and criticality of certain quality characteristics, staging pipelines can have setups that differ by quite a bit.

Tools exist for managing and orchestrating the interaction of the environments. For example, it can be ensured that only software units that have previously been verified by unit tests are integrated into the overall solution, or that complex functional tests that are not performed on the developers' local environments for performance reasons are routinely run at night.

2.3.3. Importance of unit and A/B tests in an agile environment

All test methods described in section 2.3.1 are usually also used in classical development projects. They are not used exclusively in agile development. In addition to the high degree of automation of testing, however, it is typical for agile development that unit tests and A/B tests are used intensively.

As explained in section 2.2.2, unit tests form the backbone of agile testing. In contrast, GUI tests, both manual and automated, play a subordinate role and are primarily used for random testing of previously tested modules and services across all architecture layers.

Due to the frequent delivery of releases, it makes sense in an agile environment to first test hypotheses with a small number of users using A/B tests, especially with regard to usability and new functions. Instead of speculating, as is the case with conventional fixed-price projects, about the usefulness of requirements and then including them in the specifications on a binding basis, requirements can be checked at an early stage in agile development on the basis of usage behaviour – and refined or discarded if necessary.

2.3.4. The quality assurer in the agile development team

The quality assurer (quality manager, tester or however they may be referred to) does not exist in the agile development team as a role. For example, Scrum only has the roles of Product Owner, Scrum Master and development team. The development team should be cross-functional, so that all skills for high-value and efficient software development are available in the team. Nothing discourages specially trained quality assurers from being part of the development team and perhaps even belonging to quality departments in disciplinary terms. However, it is problematic

if such a quality assurer sees themselves in the conventional tester role and considers themselves a quality policeman who is exclusively responsible for verifying defined requirements. Such an understanding would be counterproductive for an agile team. Instead, quality assurers and the entire agile team should have the following understanding:

- Agile quality assurers are part of the development team and participate on an equal footing in all meetings of the development team.
- Agile quality assurers do not play a distinct role in the team. They are not solely responsible for software quality but, as full team members, take responsibility for the entire product and its development together with all other team members.
- Agile quality assurers support the team as best they can. They can help Product Owners to identify and formulate acceptance criteria as well as assist programmers in developing unit tests and system administrators in setting up and maintaining test automation.

2.4. Quality specialists

Agile development teams are usually small units. Scrum explicitly defines the size of a development team as having three to nine members. Even if several teams are working on one instance of software, each individual team should be able to implement functional sections independently.

The agile team is responsible for the developed software, including its quality. Some quality characteristics, such as functional correctness, maturity or analysability, are inherently linked to software development. Every team should be able to consider, implement and test these characteristics. On the other hand, other characteristics require special knowledge which may not be available in the team. This includes, for example, security, performance or operating properties. The testing of security using penetration tests or the construction of performance or load test platforms in particular requires special expertise.

In such cases, Q specialists should be integrated into the development team temporarily or in an advisory capacity. It is also crucial here that the development team continues to be responsible for the entire product.

Especially in large organisations, it makes sense to establish expert pools for special quality issues and to support the development teams. As long as the technical implementation is concerned, however, it must be avoided that these expert pools develop binding standards for the teams and demand their implementation, which would weaken the autonomy of the development teams and endanger agile work. This is not the case for domain-oriented quality requirements as described in the following section.

2.5. Quality management from outside

In an agile environment, it is the development team that is responsible for the quality of the implementation of requirements. As with the conventional approach, the requirements themselves come from stakeholders. The requirements can be functional requirements, quality requirements or constraints such as legal requirements. The Product Owner is responsible for introducing these requirements into development. It is crucial that the requirements are of a purely domain-oriented nature, that is, that they do not contain any specifications on the type of implementation. Then it is also justified in agile development that stakeholders can demand the implementation of mandatory requirements and otherwise refuse to release them (e.g., in case of data privacy or functional safety requirements).

A Product Owner or stakeholder can check whether the functional requirements required by them have been implemented. Some software quality characteristics can be evaluated relatively well from the outside, such as by analysing usage behavior and testing the software. These include the quality characteristics (according to ISO 25010):

- Functional suitability: functional completeness, functional correctness, functional appropriateness
- Time behavior
- Compatibility: co-existence, interoperability
- Usability: appropriateness recognisability, learnability, operability, user error protection, user interface aesthetics, accessibility
- Reliability: maturity, availability, fault tolerance, recoverability
- Installability (if the users themselves install the software)

Although these quality characteristics are generally more difficult to validate than functional requirements, their realisation is visible and testable in principle.

The quality characteristics that are really difficult to validate are:

- Performance efficiency: resource utilisation, capacity
- Security: confidentiality, integrity, non-reputation, accountability, authenticity
- Maintainability: modularity, reusability, analysability, modifiability, testability
- Portability: adaptability, installability (if the manufacturer installs the software), replaceability

These quality characteristics are not ‘visible’ and cannot be verified from the outside by evidence; they are hidden under the bonnet. Nevertheless, stakeholders are affected by quality deficits of this kind: insufficient maintainability leads to rising development costs, lack of portability can

lead to a technological impasse, and security gaps can have considerable economic and legal consequences. A good example of this is the unintentional acceleration of Toyota vehicles: due to some untested and overly complex functions, software errors were not found and people were killed (Safety Research & Strategies, Inc. 2013).

2.5.1. The dilemma of conventional requirements engineering

Traditionally, a requirements engineer defines quality characteristics in the software specification – generally and independently of functional requirements or in relation to specific functional requirements (see, for example, Rupp et al. 2009, page 247 ff.).

Here, too, the fundamental weakness of the phase-oriented approach is evident. Many quality requirements are vaguely and imprecisely formulated in practice, and the criteria for good requirement specifications such as testability and estimability are usually not fulfilled. The effort to formulate good quality requirements is high. For example, the requirement "The software must be designed in such a way that changes can be easily implemented" is not specific at all. It is more a declaration of intent than a specific requirement. To be specific, it would have to be clarified: What does 'simple' mean, that is, what effort would be acceptable? What kind of modifications are involved? Answering these questions would require comprehensive, time-consuming analysis and specification. The effort for verifying the required quality requirements would also be high. In addition, the question arises as to whether the previously defined requirements actually meet the requirements of the users.

The specification of quality requirements which are invisible to the end user also means telling the development team how to develop the software. Such micromanagement is incompatible with agile development. In an agile environment, quality management outside the development team should rather focus on validating the resulting increment in terms of business quality criteria.

2.5.2. Field observation

Product Owners and stakeholders can observe usage behavior in the field and thus check whether and to what extent quality characteristics are fulfilled. They observe the use of the software by real users under real conditions. In an agile environment, the software is typically delivered early and updates are continuously introduced. By making observations in the field, requirements identified in this way can be incorporated into ongoing development.

As long as quality deficits can only cause minor damage, products can be inspected and hypotheses tested at an early stage of development. For example, testing alternative usage concepts by some users may in many cases be noncritical. Such A/B tests are a common tool in the development of web applications. Field observation methods can also be used for quality properties that are not immediately visible to stakeholders and users, as described below.

Performance efficiency

With the exception of time behavior, performance characteristics are not noticed by the users of the software themselves, but they can be determined by system administrators and are reflected in the cost of additional memory or computing power. These costs may be recorded and thus be an indication of inefficient resource use.

Security

Security vulnerabilities can lead to considerable damage to reputation and legal consequences. Security should therefore be validated in advance under the most realistic conditions possible (see section 2.5.3). However, even after delivery, external security experts can point out security vulnerabilities and administrators can detect and evaluate anomalies regarding possible attacks (e.g., through intrusion detection). Appropriate communication channels and monitoring measures should be established.

Maintainability

The maintainability of software consists of the characteristics modularity, reusability, analysability, modifiability and testability. All these characteristics have the same purpose: they enable efficient and thus economic ongoing development of the software during the entire product lifecycle.

As the complexity of software typically increases over time, shortcomings in maintainability lead to rising development and maintenance costs. Product Owners and stakeholders can record and evaluate these costs. Decreasing throughput – with Scrum the so-called *velocity* – is a strong indication that the software is no longer sufficiently maintainable.

Portability

The adaptability and replaceability of software can only be determined to a very limited extent in productive operation, since, for example, adaptation to another operating system or the exchange of software components are rather singular events and only a small amount of data is available about them. Nevertheless, the time required to install software can be recorded on a regular basis. Increasing installation time is an indication of lack of automation or documentation.

2.5.3. Prelive simulations

The simulation of real production conditions is an alternative or supplement to the validation of software quality in the field. This is particularly important if quality defects would lead to unacceptable damage in production operations. Simulations can be used to detect deficiencies before delivery, making them an important means of quality management both inside and outside the development team. However, Product Owners and stakeholders must avoid micromanagement and consider the purely business effects here as well.

In the following, several methods of software quality management that consider the business effects of quality defects before delivery are described.

Performance efficiency

Load tests are common methods for testing the capacity of IT systems. Typically, these tests are performed in prelive environments that are technically as close as possible to the production environment. The type and amount of data processed in the tests should also be comparable to the data in the production environment.

Security

The security of IT systems can be tested using penetration tests. In contrast to actual attacks on the production system, *white penetration tests*, that is, penetration tests in which the source code is visible to the quality assurer, can be carried out within an organisation. Penetration tests are time-consuming and are to be performed by security experts. However, this organisational separation of software development and software delivery on the one hand and penetration tests on the other should not lead to a separation of responsibilities and micromanagement. Responsibility for software security remains with the development team. Security experts advise the development team without restricting the team's autonomy through rigid security guidelines.

Analysability

While a Product Owner or stakeholder can usually only indirectly estimate maintainability properties such as modularity, reusability and modifiability over the course of development costs, analysability can be tested externally. Analysability should ensure that even new developers can understand the software architecture and source code in a timely manner. Clean documentation – such as well-written source code, comments in the source code, documented test cases or other documents – serves the purpose of making the software easily understandable and thus analysable. A new developer should be able to fix problems such as software bugs on the basis of good and appropriate documentation.

This can be tested as follows: A new developer is to try to fix a software bug. They receive the source code and all other available documents but are not allowed to talk to other team members. The developer is to speak out loud in front of the team about how they are familiarising themselves with the documentation and trying to solve the task. The developer, the team and the stakeholders then discuss the results of this test. In this way, weaknesses can be identified: lack of documentation, too much or no documentation, inconsistencies, unclear naming of methods, variables or classes or software design that is difficult to understand.

Adaptability

In many cases it is not absolutely necessary that an instance of software can run on all operating systems, browsers or end devices, but it can be a dead end if software is trapped in a specific

technical environment. Software should be developed in such a way that it can be adapted to other technologies. A Product Owner can check this through initiation of the adaptation of a small part of the software to another technology at an early stage and recording the effort involved. If the effort is high, this is an indication of a lack of adaptability. In this case the relevant reasons should be discussed with the development team and measures to improve the flexibility of the software design should be defined.

2.6. Excursus: Documentation

The Agile Manifesto states: "Working software over comprehensive documentation." This agile value in particular seems to be misunderstood. The opinion of a functional safety engineer at the Automotive SYS 2017: "Agile development cannot work, because nothing is documented."

The agile principle is that functioning software and not documentation should be the focus of software development, but this does not mean that documentation is obsolete in an agile environment. Documentation is also part of the agile development process, but the team should avoid documentation for its own sake.

The agile team is responsible for the developed software and the type of software development. In this way the team also decides on the type and scope of the documentation. The team is responsible for the quality and thus the maintainability of the software as well, which must be ensured by adequate documentation. Nevertheless, there are also documents that are externally required, the creation of which is the responsibility of the Product Owner.

The following describes which documentation types exist in the context of software development and how they should be handled in an agile environment.

2.6.1. Documentation of software development

In conventional phase-oriented software development, there are a number of documents that may be designated for the development of the software, including requirement specifications, test plans and analysis reports. For example, Automotive SPICE alone proposes no less than 19 output work products for the *Software Engineering Process Group* (SWE), 14 of which are document types primarily intended for software development (A-SPICE 2017).

Scrum, on the other hand, provides only three artifacts for software development besides the increment itself: the product backlog, the sprint backlog and the definition of done. The development team also decides which artifacts are created. The crucial factor is the working increment at

the end of a sprint, which is precisely what the agile principle “Working software over comprehensive documentation” refers to.

2.6.2. Documentation for release

In an agile environment, there is ideally no formal release process that would require additional documentation. Instead, the Product Owner, who is part of the agile team and usually works closely with the development team every day, decides when a finished increment is delivered.

Nevertheless, external constraints may require documentation for release in practice. For example, evidence that documents the protection of the software against tampering could be required by the authorities. Such documentation can hinder agile work, but in most cases agility should still be possible.

In such cases, it is the task of the Product Owner to monitor the corresponding requirements in the development process and to ensure their implementation before delivery of the software. This requires processes outside the actual agile work and agile teams to ensure that all relevant requirements are known to the Product Owner.

2.6.3. Software maintenance documentation

The documentation for software maintenance must ensure that a software module can still be analysed and understood even if the module has not been worked on for a long time and the developers of that time are no longer available to the team. During software development, it must therefore be checked whether maintainability is sufficiently guaranteed (see “Analysability” in section 2.5.3).

It should be taken into account that all kinds of artifacts can be used for documentation. In addition to the source code, this also includes automated test cases, which should be available in large numbers in an agile environment.[4]

2.6.4. Documentation as part of the product

Documents, such as user manuals, may be intended for the user of the software. Such documents are part of the product and must be treated as such in agile development. For example, the Product Owner could formulate a PBI (“As a user, I would like to have a user manual available for the

[4] Traditionally, the documentation of the software consists of separate system documentation. In the author’s experience, such system documentation often does not correspond to the current development status of the software due to high maintenance costs. Out-of-date system documentation often raises more questions than the answers it provides. In contrast, source code and test cases that are continuously tested as regression tests are always up-to-date.

software to become familiar with how the software works") and introduce it into the development process. To ensure continuous maintenance of the user manual, the Definition-of-Done is then extended so that a requirement is not successfully fulfilled until the user manual has been updated.

However, the following also applies here: Documentation for the user is not an end in itself. The goal is that the user can learn how to use the software. There are alternatives to the classic user manual that should be considered.[5]

2.7. Process quality

Up to now, the quality of the created software has been in the foreground. However, quality management traditionally also includes defining and reviewing requirements for process quality.

Within the meaning of the agile principle "individuals and interactions over processes and tools", the creation of the development process by the agile team is more important in agile development than external specification of process instructions. Agile teams typically work according to an agile process framework such as Scrum. The agile team selects and shapes the specific development methods within this framework primarily by itself and continuously validates and optimises the methods through regular retrospectives, for example.

The Team Agility Master (*Scrum Master* with Scrum) plays an important role here, moderating retrospectives and providing significant support to the team in overcoming impediments within and in cooperation with other organisational units. In this way the Team Agility Master decisively shares co-responsibility for the quality of the development process. As in conventional development, they can also introduce and validate process requirements. As is usual in an agile environment, this is not done through one-off formal instructions but through dialogue and continuous cooperation with the development team and the Product Owner.

2.8. Conclusion

Agile software development supports the development of high-quality software in many respects: through customer feedback at an early stage, potential for improvement, particularly with regard to functional suitability and usability, can be recognised and implemented early on; frequent

[5] For example, most game apps familiarise users with how to use the app in a playful manner. Reading manuals is unusual even for complex games.

integration and delivery of the software improves maintainability and installability; the quality of the development is the responsibility of the autonomous team and thus of the developers who know the software best. However, this also means that the quality of the software depends decisively on the quality and thus on the motivation and competence of the team, especially the development team. Quality management in an agile environment is, if nothing else, talent management.

Just as agile software development has not made requirements engineering obsolete but has fundamentally changed it, quality management has also been changed. Quality management no longer serves as a control instance between actual development and delivery but instead takes place mainly in the development team itself and through the observation of usage behavior in the field.

Today, modern tools for automated testing, integration and delivery of software enable the development team itself to deliver software in a quality-assured manner and with very little manual testing effort. Explicit test competence by experienced quality assurers is still valuable in an agile team, but the role of quality assurer is changing: instead of being a quality policeman, they are becoming full members of the development team, supporting all activities of the team with their special skills.

The agile development team may be supported by external quality experts, but the responsibility for product quality remains with the agile team.

The Product Owner and external stakeholders impose requirements in regard to the software, also with regard to quality. However, they respect the autonomy of the development team with regard to technical implementation and do not engage in micromanagement. Rather, they focus on how the product is used in the market in a safe and reasonable manner. This closes the important control loop that allows experience gained in the field to be incorporated into ongoing development.

3. SOFTWAREDEVELOPMENT IN THE AUTOMOTIVE INDUSTRY

The automotive industry does not develop software fundamentally differently from other industries. The electrons used to process data in an engine control unit are identical to the electrons used by social media platforms or game apps. Nevertheless, depending on the type of product, special aspects must be taken into account that distinguish the development of digital products in the automotive industry from the typical agile development of web applications:

- Embedded systems, such as control units in a vehicle, are developed for special, clearly defined applications. This specific purpose relates not only to the software but also to the hardware. Hardware and software development take place hand in hand and are part of system development. Application development, on the other hand, is based on existing hardware (smartphones, servers, etc.) including the corresponding operating systems and basic software. Such software development is independent of hardware development.
- A vehicle is a safety-critical system; a web application or an app is usually not. The requirements for a safety-critical component such as an engine control unit or a parking assistant are far more demanding in terms of performance and maturity than for most web applications or apps.
- For the type approval of vehicles, legal requirements must be fulfilled which also apply to IT systems and their development processes. In addition, there are industry-specific standards and recommendations that must be met. These include Automotive SPICE and ISO 26262, which have a direct influence on the development of control units for vehicles and must also be taken into account in an agile environment.
- Almost all IT systems in vehicles are developed by suppliers. The OEMs, who are responsible for the entire vehicle product in regard to the end customers and authorities, must vouch for the quality of important IT components without having developed them and without having detailed insight into them. This is a particular challenge for quality management in the automotive industry.

Today, almost all apps, web applications and desktop programs are developed agilely, while embedded systems are rarely developed agilely in the automobile industry. However, due to increasing digitalisation of products and services in the automotive industry, the digital portfolio of

OEMs has grown considerably: the infotainment system is already one of the most comprehensive and complex IT systems in the vehicle, vehicles are increasingly networked with the outside world (V2X) and access digital services outside the vehicle, and OEMs are developing and operating web platforms and apps as communication media for their customers. As a result of this expansion of the digital product range, agile development approaches have now found their way into the automotive industry. Depending on the criticality of the IT system, however, it must also be investigated whether and to what extent laws and guidelines regarding product safety are to be taken into account and whether agile development methods are compatible with them.

Due to this product diversity in the automotive industry, the question of whether and how IT systems can or should be developed in an agile manner cannot be answered in general terms. However, there are industry standards that must be taken into account in many developments. They include:

- Automotive SPICE
- ISO 26262
- AUTOSAR
- Maturity level assurance for new parts according to VDA

Due to the great importance of these four standards in the automotive industry, they are briefly described below and their compatibility with agility is discussed.

3.1. Automotive SPICE

3.1.1. Brief description

Automotive SPICE is an automotive industry standard for assessing the maturity of development processes at manufacturers of embedded systems. It is a domain-specific variant of the ISO/IEC 33020 (SPICE) standard. Automotive SPICE defines the *Process Reference Model* (PRM) and the *Process Assessment Model* (PAM) for development of embedded systems. It adopts the measurement framework (capability levels, process attributes, rating, process capability level model) of SPICE unchanged (see A-SPICE 2017). Automotive SPICE is currently available in version 3.1 from November 2017.

Automotive SPICE structures the processes according to the V-model (A-SPICE 2017, p. 122). In Automotive SPICE, the V-model refers both to the superordinate process of system development and to the subordinate process of software development. The development processes for hardware and mechanical systems are not considered in Automotive SPICE.

The *Process Performance Indicators* defined in the PAM, consisting of the *Base Practices* (BP) and the *Work Products* (WP), are merely indicators and therefore not binding rules for the implementation of Automotive SPICE.

3.1.2. Automotive SPICE and agility

A process maturity model such as Automotive SPICE only describes the WHAT, that is, the goals of the process. In contrast, a process model such as Scrum describes the HOW, that is, with which methods, means and organisational structures the objectives are to be achieved (see A-SPICE 2017, p. 23). The *Automotive SPICE Guideline* therefore expressly states that Automotive SPICE is compatible with agile frameworks (VDA 2017, Chapter 2.2.2). A summary of the compatibility of Automotive SPICE and agility can also be found in the whitepaper *Clarifying Myths with Process Maturity Models vs. Agile* (Besemer et al. 2014), in *How to tell the assessor?* (Bondar et al. 2015) and in *How Do Agile Practices Support Automotive SPICE Compliance?* (Diebold et al. 2017).

Relation of the V-model

Besemer et al. write that SPICE, and thus Automotive SPICE as well, does not specify the order in which the individual objectives are to be achieved: "Since SPICE is at the WHAT level it cannot, and does not, predefine any lifecycle model. Consequently, it does not predefine any logical points in time at which work products shall be available, nor does it predefine any other kind of activity sequences or ordering." (Besemer et al. 2014, p. 6). This means that neither the V-model nor Automotive SPICE explicitly stipulate that the V-model is a phase model for the entire development period. The V-model can also relate to a release, a sprint, or the implementation of a single requirement. If many small V-processes, each of which refers to a single requirement, are run through during the entire term, the fundamental conflict with agility is resolved.[6] This is crucial to enabling compatibility between Automotive SPICE and agility. The typical interpretation of the V-model as a model that provides for phases to be processed sequentially in relation to a project or a release would be incompatible with an agile approach and is not required in Automotive SPICE either. However, the relation of the V-model to the realisation of individual requirements enables agile development.

Chronological independence of process outcomes

A further prerequisite for efficient agile development of systems is chronological independence of the individual process outcomes within a single process. This is necessary both to decouple process steps for strategy development from operational tasks and to enable test-driven development.

[6] One could also relate the V-model to an iteration – the sprint according to Scrum. This would correspond to a mini-waterfall model, but it is not truly agile and should be avoided.

For example, SWE.4 *Software Unit Verification* states: Process outcome 1 requires "a software unit verification strategy including regression strategy […] developed to verify the software units." The actual software unit verification has to be carried out in the same process. This blending of strategic and operational tasks within a process suggests that these process results are developed in a timely manner within a development phase. This makes sense with the classic waterfall model: After creating all software units, a strategy for their verification is created and then the verification of all software units is carried out. This would not make sense with agile development. Accordingly, a strategy would have to be developed for each verification of an individual requirement. In fact, process outcome 1 should be interpreted in such a way that it is worked out in advance once for all software units and then the individual software units are continuously tested accordingly during development. This argumentation applies analogously to the process outcomes SYS.4: 1 and 2, SYS.5: 1 and 2, SWE.5: 1 and 2, SWE.6: 1, SUP.1: 1 and SUP.2: 1.

In addition, for process SWE.4, process outcome 2 ("criteria for software unit verification are developed [...]") should be understood independently of the actual verification of the software unit. For the purpose of test-driven development, test cases are described before realisation of the software unit. This also applies to the process results SYS.4: 3, SWE.5: 3 and SWE.6: 2.

The processes defined in Automotive SPICE are merely thematic bundles. The chronological order of the processes or their process outcomes must not be taken into account when assessing the degree of maturity.

Individual process outcomes with conflict potential

In addition to these fundamental interpretations of Automotive SPICE, there are several process outcomes that need to be clarified so that they do not impede agile system development. These include, but are not limited to:

- ACQ.3, process outcome 2: "the contract/agreement clearly and unambiguously specifies the expectations, responsibilities, work product/deliverables and liabilities of both the supplier(s) and the acquirer" – This process outcome is certainly also useful and necessary for agile system development, but it should not be understood in such a way that a complete requirement specification is required as an integral part of the contract. This would be contrary to the agile approach. Instead, it must be possible to design contracts that are compatible with agility (see section 1.5).
- SYS.3: "The purpose of the System Architectural Design Process is to establish a system architectural design and identify which system requirements are to be allocated to which elements of the system, and to evaluate the system architectural design against defined criteria." – The agile development team breaks down the business requirements into technical requirements only when needed, thus iteratively defining the architecture. Architectural decisions are made more ad hoc and less formal. This also applies to processes SWE.2 and SWE.3.

- SWE.3, process outcome 5: "results of the unit verification are summarised and communicated to all affected parties" – The implementation of the software and its unit tests, the verification on the unit level and checking in of the source code takes place in agile development in short cycles and autonomously by the developer (see section 2.3.1). This may involve team members who may then carry out further tests. Documented summaries and communication are not mandatory. Rather, the individual test cases in the agile environment are usually recorded in the form of automated tests and thus individually documented.

- SUP.1, process outcome 2: "quality assurance is performed independently and objectively without conflicts of interest" – Quality assurance is the duty of the entire agile team. A dedicated role for quality assurance is not foreseen in Scrum and other frameworks. It is common practice for a person other than the developer who implemented a requirement to test the integrated result before the Product Owner reviews and approves the increment themselves. However, the required independence of quality assurance applies not only to the product but also to the processes. In an agile environment, processes are reviewed and improved from within the entire team, typically through retrospectives. Independence would at best be guaranteed by the independent Agility Master.

- SUP.9, process outcome 2: "problems are recorded, uniquely identified and classified" – Automotive SPICE does not make a clear statement about what kind of problems these are. When software bugs of software that has already neen delivered are involved, this process outcome also makes sense in an agile environment. However, if the bugs are detected during development, the process step is too formal and cumbersome. If a quality assurer detects a bug and immediately talks to the developer to fix it, no documentation should be required. This is common practice in an agile development team working at one site.

- SUP.10: "The purpose of the Change Request Management Process is to ensure that change requests are managed, tracked and implemented." – Requirements management in an agile environment is already based on continuous changing of requirements. Change Request Management is implicitly already available and an additional explicit process is not necessary in an agile context.

Process performance indicators

Automotive SPICE supplements the PRM with the PAM, which includes the process performance indicators *base practices* and *work products*. However, some of these indicators are not useful for agile development and violate in particular the agile principles of not specifying processes in too much detail and restricting documentation to the necessary extent. For example, the detailed design of software units within the framework of Scrum developments is discussed and realised by the development team during the sprint planning or sprint. Comprehensive documentation, as described in the work products of process SWE.3 – for example, the *review record*, in which, among other things, the reviewers are listed and the status of the review and the time required for

the review are stated (see A-SPICE 2017, p. 109) – would be completely inappropriate in an agile environment. However, the process performance indicators are merely indicators and not binding criteria for assessing process capability: "Therefore, work product & characteristics are neither a 'strict must' nor are they normative for organisations. Instead, the actual structure, form and content of work products and documents must be decided by the organisation ensuring that it is appropriate for the intended purpose and needs of its projects and product development goals." (Besemer et al. 2014, p. 7).

Nevertheless, critical questions must be asked regarding whether these indicators, even if they are formally non-binding, **in practice** do not have a considerable influence on the assessment of process capability. For example, a development manager who ignores indicators and tries to achieve the process result by other means may have to justify this decision if the development project fails; however, if all base practices are meticulously adhered to and all work products created, failure is more likely to be accepted as inevitable. Likewise, a company can hardly be sure that an assessor, as actually required by SPICE, will in fact only use the PRM as a binding yardstick and not the indicators.

It would therefore be desirable to supplement base practices and work products with explanations that better illustrate their scope and thus their compatibility with agile methods. There are already initial approaches to this: Diebold et al. in their paper *How Do Agile Practices Support Automotive SPICE Compliance?* (Diebold et al. 2017) mapped agile practices to base practices and work products, covering 96% of base practices and 87% of work practices.

Capabillity levels

SPICE defines a measurement framework with capability levels from CL 0 to CL 5. Provided that the Automotive SPICE Process Reference Model can be interpreted – as described above – for the purpose of agile development, the question regarding which capability levels agile teams can reach arises.

Level CL 0 (*incomplete process*) means that the process does not exist or that the required process outcomes are not achieved.

For level CL 1 (*performed process*), the process outcomes must be achieved no matter how. Agile teams can certainly achieve CL 1 under the above conditions.

CL 2 (*managed process*) requires that the process is planned, monitored and controlled. Automotive SPICE requires as a minimum that resources, effort and time are managed. In principle, this can also be done in an agile manner. Agile frameworks like Scrum explicitly define rituals, artifacts and roles. However, additional measures may be required to ensure traceability of work products (A-SPICE 2017, PA 2.2.). Typically, many agile teams work with index cards attached to walls. There is no traceability under these circumstances. Depending on the work product, it is necessary to clarify in individual cases whether the work outcome is documented or whether traceability must be guaranteed. However, the use of software systems for requirements and version management that support traceability is also common in agile environments. Agile

development teams typically create many unit test cases that reference requirements and are managed together with the source code in a version management system. The required traceability should therefore not be a fundamental impediment to agility.

CL 3 (*established process*) demands a standard process for development of a product type that is valid for all projects and teams. If this required process standard also refers to the work within the agile team, this is a fundamentally different approach than in agile software development and therefore incompatible with the agile principles. Although there are agile frameworks such as Scrum or Extreme Programming that define a procedural framework, the goal is for each development team to define and continuously adapt a process that makes sense for their own needs in coordination with organisational units to which they have connections. CL 3, on the other hand, calls for a centrally managed process that is universally valid and can only be adapted by tailoring. In an agile environment, CL 3 can only be achieved if the required standard process is very universal and primarily targets the interfaces between teams and organisational units. Otherwise, the self-organisation that is so important in agile development would not be feasible.

CL 4 (*predictable process*) calls for the establishment of a key performance indicator system in order to be able to evaluate quantitatively the extent to which the business objectives are achieved by the development process and what optimisation potential exists. Such a focus on business objectives is certainly in the interest of agile development (see also section 1.6). Likewise unproblematic are key performance indicators that measure the amount of implemented requirements (e.g., velocity for Scrum). However, it would be problematic if intermediate results and individual competences were measured and evaluated within an agile team. In an agile environment, the key performance indicators should be limited to the software created, its use in the market and the efforts and turnaround times of the entire team. Detailed controlling within the team should be avoided.

The process requirements added for the final capability level CL 5 (*innovating process*) fit very well with agile values and principles. Agility includes continuous evaluation and adaptation of the development processes by the team. Due to the high degree of self-organisation of the agile team, ideas for product and process improvements can be introduced and tested without significant organisational effort.

3.1.3. Conclusion

To develop systems both according to Automotive SPICE and according to agile principles, Automotive SPICE must be interpreted differently than is usual for conventional development projects. This concerns the following aspects in particular:

- The V-model refers to individual requirements, not to the overall project.
- Process areas and their outcomes must be considered chronologically independent. Strategic tasks are therefore chronologically independent of operational tasks.

- Quality assurance measures within the agile team, such as mutual reviews, pair programming or retrospectives, must also be allowed as an independent quality management system. External quality management must not interfere with the internal processes of the agile team.
- Process performance indicators should only be understood – in the sense of SPICE – as indicators and not as a binding yardstick for process assessment.

With regard to capability levels, achieving CL 3 in an agile environment is difficult, as the agile approach deliberately dispenses with general process standards and instead the agile team develops and continuously adapts its own processes and standards within the agile framework.

The question is not so much "Is agility compatible with Automotive SPICE?" (The answer is yes, in principle – at least until maturity level 2.), but "Is the interpretation of Automotive SPICE necessary for agile development accepted by the assessors?" The *International Assessor Certification Scheme* (INTACS) is an important advocate for the compatibility of agility with Automotive SPICE, so that a fundamental willingness to accept agile methods as a substitute for conventional project management methods can be assumed.

3.2. Functional safety (ISO 26262)

3.2.1. Brief description

ISO 26262 ("Road vehicles – Functional safety") is an international standard for safety-relevant electrical and electronic systems in vehicles. ISO 26262 came into force in November 2011. The following considerations refer to the draft status of 2018 (ISO/FDIS 2018).

The aim of ISO 26262 is to define requirements and processes in order to get systematic errors and random hardware errors under control:

"[…]

a) provides a reference for the automotive safety lifecycle and supports the tailoring of the activities to be performed during the lifecycle phases, that is, development, production, operation, service and decommissioning;

b) provides an automotive-specific risk-based approach to determine integrity levels [Automotive Safety Integrity Levels (ASIL)];

c) uses ASILs to specify which of the requirements of ISO 26262 are applicable to avoid unreasonable residual risk;

d) provides requirements for functional safety management, verification, validation and confirmation measures; and

provides requirements for relations with suppliers." (ISO/FDIS 2018:8)

Like Automotive SPICE, ISO 26262 is based on the V-model. In contrast to Automotive SPICE, hardware development is explicitly addressed as part of system development in addition to software development; as with Automotive SPICE, the development of mechanical systems is not taken into account. ISO 26262 describes the development process along the V-model in parts 3 ("Concept phase"), 4 ("Product development at the system level"), 5 ("Product development at the hardware level") and 6 ("Product development at the software level"). Part 7 ("Production, operation, service and decommissioning") deals with the product lifecycle after development. Parts 3 to 7 describe process specifications[7] for quality assurance. The individual process specifications described there, especially in Part 6, are mostly specific neither to the automotive industry nor to safety-critical systems. Rather, they are generally specifications that can and are applied meaningfully in other industries and to non-critical systems.

ISO 26262 compliant development has three essential characteristics:

1. **Production of work products[8]** that make it possible to demonstrate functional safety. These artifacts build on each other in part.
2. **Examination of these work products** to ensure that the basis for the final safety demonstration is correct.
3. **Compliance with process specifications** during development, depending on the classification of the system to be developed.

At the heart of the work products is the *Hazard Analysis and Risk Assessment* (HARA), which must be performed prior to system development. The result of the HARA is the classification of the safety criticality of the system to be developed into one of the *Automotive Safety Integrity Levels* (ASIL). Depending on the classification in QM or ASIL A to ASIL D, different requirements arise with regard to the quality assurance measures described in Parts 3 to 7 for the verification or validation of the product and other work products. In addition, safety analyses, e.g. in terms of a *Failure Mode and Effects Analysis* (FMEA), are to be carried out during system development (ISO/FDIS-9:2018, Section 8).

ISO 26262 pays particular attention to the testing of work products (e.g., the HARA). These tests are referred to in ISO 26262 as *confirmation measures*. Table 1 of part 2 of ISO 26262 lists the necessary confirmation measures and describes the requirements for these tests. A crucial aspect

[7] Here and in the following, *process instructions* should be understood as all activities for the implementation of the functional safety requirements. This includes measures to ensure functional safety but also activities related to the safety culture in the company.

[8] Here and in the following entire section on functional safety, *work products* always refer to the documents or other work products demanded by ISO 26262.

here is the relationship between the auditor and the person responsible for creating the work product. ISO 26262 distinguishes between four degrees of independencey:

Degree of Independency	Requirement for inspection
I0	The confirmation measure **should** be performed; however, if the confirmation measure is performed, it shall be performed by a **different person** in relation to the person(s) responsible for the production of the work product.
I1	The confirmation measure **shall** be performed, by a **different person** in relation to the person(s) responsible for the production of the work product.
I2	The confirmation measure **shall** be performed, by a **person from a different team**, i.e. not reporting to the same direct superior.
I3	The confirmation measure **shall** be performed, by a **person from a different department or organization**, i.e. independent from the department responsible for the considered work product(s) regarding management, resources and release authority.

Higher ASILs have the same or higher independence requirements than lower ASILs. There are work products that need to be checked for all ASIL classifications with the highest degree of independence (e.g. HARA).

Depending on the ASIL defined in the HARA, certain process specifications must be considered during the development of a system. Process specifications exist for all development phases (development of the entire system, development of the hardware, development of the software).

ISO 26262 defines the following roles and responsibilities:

- The *Project Manager* is responsible for ensuring that the safety goals for the overall project are achieved and that the necessary resources are provided.
- The *Safety Manager* is responsible for planning and implementation of safety activities and for compliance with and adaptation to the safety plan (ISO/FDIS 26262-2:2018, 6.4.6). They may delegate responsibilities provided they are clearly assigned and communicated (ISO/FDIS 26262-2:2018, 6.4.6). A Project Manager can also be a Safety Manager (ISO/FDIS 26262-2:2018, 6.4.2.4 Note 1).

3.2.2. ISO 26262 and agility

ISO 26262 explicitly permits the use of agile methods also for the development of safety-critical software (ISO/FID 26262-6:2018, 5.2, Note 1), provided that the required documentation obligations and implementation of the required safety processes are ensured. This results in four fundamental sources of conflict potential between the standard and agility:

Process model

Similar to Automotive SPICE, ISO 26262 is based on a V-model. In contrast to Automotive SPICE, ISO 26262 specifies a chronological sequence of the process steps, in which prerequisites are required for the start of a process step, which must be worked out beforehand by other process steps. However, it can also be argued here that the V-model does not refer to the entire development cycle but to smaller units down to the implementation of individual requirements. That means the same argumentation regarding the compatibility of V-model and agility applies as for Automotive SPICE (see section 3.1.2).

Nevertheless, there are safety-relevant activities which, according to ISO 26262, must be carried out at least at the beginning of a project. This includes HARA in particular. Depending on the types of changes that occur in the course of development, HARA must be repeated several times and ASIL may need to be changed, or HARA is only carried out once at the beginning if the changes have only a minor impact on functional safety. Independently of this, the (first) HARA should be carried out in an initial concept phase.

Documentation

ISO 26262 requires generation of a series of documents as evidence of development in conformity with the standard. The creation of these documents by the team can be understood as requirements for the product to be created (see also section 2.6). The team can incorporate these constraints into the agile development process in the form of definition of ready, product backlog items and definition of done. Therefore, the required extent of documentation does not contradict the agility in terms of processes, even if it means increased maintenance and modification effort.

Nevertheless, the documents required by the standard contradict the Agile Manifesto, according to which working software is to be valued more than comprehensive documentation: The documents required by the standard are primarily aimed at documenting product quality and are therefore not part of the product itself. The more extensive the documentation effort (depending on the ASIL), the more ISO 26262 contradicts agility with regard to the documentation requirements and thus impedes agile development.

Method competence and application

In Parts 3 to 6, ISO 26262 describes methods for testing functional safety and recommends these methods with varying degrees of urgency, depending on the ASIL. The methods generally do not differ from other common methods for verification and validation of software and are also applicable and common for agile development. Good software developers and software quality assurers should know and master the testing methods recommended in ISO 26262, regardless of whether they work in a conventional or agile environment.

For agile work, however, it must be critically considered that these methods are prescribed externally by the standard. In accordance with the agile values and principles, the development team should be able to decide for itself which methods to use and to what extent.

The agile team must be able to produce the work products required by ISO 26262 and to identify indicators, e.g. for necessary adaptation of the safety plan or for renewed execution of HARA. These skills must be demonstrated in accordance with the standard. In addition, the ISO 26262 roles of *Project Manager* and *Safety Manager* must be established in the agile team. For example, the Product Owner could assume the role of Project Manager and the Agility Master the role of Security Manager. These requirements are basically compatible with an agile approach, but require explicit and verifiable expertise regarding functional safety in an agile team. If necessary, functional safety experts must support the team, including the Product Owner or the Agility Master, in performing these tasks.

Independence of the confirmation measures

ISO 26262 requires that conformation measures be carried out and that different requirements regarding the organisational independence of the auditors be met, depending on the ASIL (see ISO/FDIS 26262-2:2018, table 1). On the other hand, the agile principle is that the development team develops the shippable increments on an autonomous and self-organised basis.

The critical question is whether or not the auditor, that is, the person responsible for the confirmation measures, is part of the development team. If this is the case, even then an explicit role would be defined within the agile team, which would be a violation of the agile principle of joint responsibility of the development team for the product, including functional safety. However, if the auditor is not allowed to be part of the team, the contradiction between the standard and agility becomes even more pronounced.

The more safety-critical the product to be developed is (according to the ASIL classification), the higher the general requirements for auditor independency and the more challenging it is in many organisations to continuously involve independent auditors in the agile development process.

ISO 26262 follows a fundamentally different strategy than the approach of self-organisation in agile development with the independent examination of safety-relevant work products: The standard relies on independent control, while agility relies on the responsibility of the development team. It cannot and should not be decided at this stage which of these two approaches is more effective and better ensures functional safety. There are good arguments for both approaches (for the self-organising approach, see also section 2.2.2).

Conclusion

A general statement on the compatibility of ISO 26262 with agile development cannot be made. A possible scenario of how such an integration of ISO 26262 into an agile team could look like is described by Prof. Vogelsang of TU Berlin in appendix B.

The approaches of ISO 262626 differ from those in the agile environment both with regard to the documentation effort and the methodological requirements for the development team as well as the organisational independence from creators and auditors of security-relevant work products.

How well ISO 26262 can nevertheless be combined with agility also depends, among other things, on

- how closely the development team and the auditors of the safety-relevant work products can work together in the organisation,
- how efficiently the development team can produce the necessary documentation,
- to what extent functional safety competencies are available in the agile team and
- how safety-critical the product to be developed is.

Irrespective of this, an initial concept phase in which a (first) HARA is developed for the system is likely to be both reasonable and unavoidable, even in the case of agile development.

3.3. AUTOSAR

3.3.1. Brief description

The development cooperation AUTOSAR (**AU**T**O**motive **O**pen **S**ystem **AR**chitecture), founded in 2003 by companies in the automotive industry, has defined a standardised software architecture for ECUs. This architecture ensures consistent separation of hardware-independent software components (SWCs) and the infrastructure and services implemented by basic software (BSW). This decoupling enables software applications to be implemented and tested independently of the underlying hardware. The AUTOSAR runtime environment (RTE) controls the connection of the SWCs to the basic software. Data is exchanged between SWCs using the Virtual Functional Bus (VFB).

The entire system is defined by an AUTOSAR XML file that describes the SWCs and determines their distribution to the individual ECUs and network communication. *ECU Extract of System Descriptions* can be generated from this overall description, which then serve as the basis for developing individual ECUs. Exchange of system descriptions between the development partners can therefore essentially take place via XML files.

3.3.2. AUTOSAR and agility

In accordance with agile values and principles, it is up to the development team to decide how to implement the technical requirements. AUTOSAR as a technical platform and reference architecture is only one possibility of implementation. If the entire software is developed autonomously by a single development team – which would not be typical in the automotive

industry – the agile team should be able to decide for itself whether development should be based on AUTOSAR or not.

In fact, the vast majority of developments are likely to be distributed across different teams, and AUTOSAR may be specified as the reference architecture. In this case, the technical requirements resulting from AUTOSAR represent boundary conditions that are incorporated as requirements into the development process and must be taken into account. Only in this way can the services and infrastructure provided by AUTOSAR be used. This also does not impede agile development of the software.

The description of requirements according to the AUTOSAR XML files also does not exclude agile development. These XML files can be part of the product backlog items or referenced in the user stories.

3.3.3. Conclusion

AUTOSAR describes a technical framework for the development of ECUs in the automotive industry, not a development model or procedure. This framework can be used independently of the development paradigm. Therefore, AUTOSAR and agility do not contradict each other.

3.4. Maturity level assurance for new parts according to VDA

3.4.1. Brief description

The VDA Blue Gold Volume *Product creation – Maturity level assurance for new parts* defines measurement criteria for a total of eight maturity assessments (VDA 2009). These eight maturity assessments build on each other sequentially and thus define a phase model for the development of new parts. Depending on the classification of the new parts to be developed into one of the maturity level risks A, B or C, different processes for the cooperation between the customer and the supplier for evaluation and discussion are described: starting with the transmission of the results of the self-evaluation by the supplier to the customer (risk level C), to the presentation of the self-evaluation results at a meeting (risk level B), through to the joint maturity level assessment by the customer and supplier at a round table (risk level A).

Even though the standard claims to apply to the development of software as well (VDA 2009, p. 16), its origin is clearly in the development of mechanical components, e.g. in terms of the requirements for timely availability of production facilities, concepts for packaging, etc.

3.4.2. Maturity level assurance and agility

The maturity level assurance for new parts proposed by the VDA describes a phase model for product development and thus demands a waterfall model, transferred to software development. This approach is therefore fundamentally incompatible with agility:

The first maturity level ML0 *Innovation release for full production development* is used for the release of funds by the customer and is still independent of the development method. ML1 *Requirements management for the contract to be issued* already includes release of a requirement specification (VDA 2009, p. 43 ff). This might be reinterpreted for agile development and understood as an initial product backlog. However, for ML2 *Specifying the supply chain and placing the order*, completeness of the requirement specification (VDA 2009, p. 49 ff) and for ML3 *Release of technical specifications*, completeness of the technical specification (VDA 2009, p. 58 ff) is required. At the latest with ML3, the level of the maturity process cannot be reconciled with agile principles even through generous reinterpretations of the required artefacts. The evaluations ML4 *Completion of production planning* and ML5 *Parts from production tools and production facilities are available* are not fundamental contradictions to agile development. The last two evaluations ML6 *Process and product approval* and ML7 *Project completion, transfer of responsibility to production, start requalification* reflect the project character with clear delivery time of the final result. This is rather atypical in an agile environment but does not fundamentally hinder agile development and is necessary in the context of overall vehicle development.

The maturity level assessments described by the VDA are an important instrument in many companies in the automotive industry for evaluating the entire development status up to the start of production (SOP). In order to enable agile development of new parts into a conventional perspective of overall development according to the phase model described above, the phase model would have to be changed in some places. One solution could be, for example, that for agile developments

- an initial product backlog with all the mandatory requirements known at the time of creation must be created instead of a comprehensive requirement specification (concerns the maturity level assessments ML1 and ML2),
- a technical specification is not required, that is, new parts developed in an agile manner are not taken into account in the ML3 maturity level assessment,
- ML4 maturity level assessment demands a concept for DevOps and their implementation but not the started or even completed development of the new part and
- ML5 maturity level assessment only requires implementation and delivery of mandatory requirements and proof of successful integration into the overall system.

Such a change would provide a framework in which new parts developed in an agile manner would be taken into account in the overall evaluation as far as possible without substantially violating agile principles.

3.4.3. Conclusion

The maturity level assurance for new parts according to VDA is well suited for the preparation of series production of mechanical parts, only conditionally for software development, not for agile development and certainly not for agile software development. In particular, the requirements for maturity level 3 *Release of technical specifications* is incompatible with agility. Scopes developed in an agile manner should not be taken into account in maturity level assessment ML3.

4. QM OF AGILE SOFTWARE DEVELOPMENT IN THE AUTOMOTIVE INDUSTRY

In the following, the challenges and solutions for quality management in the automotive industry in the introduction of agile development methods for digital products are presented. The entire spectrum of possible digital products in the industry in general, as well as developments typical for the industry, such as safety-critical control devices in particular, are considered. The starting point for this is the familiar tasks of quality management:

1. **Define and introduce quality requirements**: Capturing and formulating quality requirements with regard to products and methods and steering them into the development process
2. **Evaluate suppliers**: Audit suppliers and check their ability to implement requirements
3. **Assure maturity level**: Verify and validate product and process quality
4. **Release products**: Final check of product and release for delivery
5. **Observe the field**: Record product usage by the customer and complaints
6. **Derive lessons learned**: Record findings from field observation and the development and production process and integrate them into the developments

For each of these tasks, it will be shown how quality management can be reasonably implemented in agile development of digital products.

4.1. Define and introduce Q-requirements

4.1.1. Product classification and Q-requirements

Depending on the type of product, different subject-specific quality requirements have to be implemented as well as organisational and legal constraints and industry standards have to be taken into account. The first step therefore should be to clarify the class to which the product to

be developed belongs and the requirements associated with the product class before development begins.

Example parking assistance system [9]

A new parking assistance system that enables automated parking even in more complex parking situations (garages, multi-storey car parks, etc.) is to be developed. This results in a number of technical quality requirements, such as with regard to the maintainability of such a system by workshops and reliability. Many of these requirements are already known from previous developments of parking assistance systems and can be used again for the new development. The parking assistance system is an embedded system that communicates with other ECUs in the vehicle via the CAN bus and is based on the AUTOSAR platform and its architecture. The system must be implemented according to the requirements of Automotive SPICE. It is also a safety-critical system that must therefore be developed in accordance with ISO 26262. It is assumed here that the hazard and risk analysis ASIL D, that is, the highest safety-critical level, applies[10].

Example parcel delivery

In the future, parcel delivery companies will be able to deliver parcels to the boot of vehicles. Several systems must be developed or adapted for this purpose:

1. In online shops, customers can enter the registration number of their vehicle as the delivery address, the two-hour time slot for the desired delivery and the location of the vehicle during this time slot.
2. The parcel delivery system must be expanded to include the registration number, time window and location and must be able to receive this data from online shops.
3. The vehicle must transmit a GPS signal with the exact location of the vehicle to the parcel deliverer.
4. The parcel delivery system must receive and process the GPS signal.
5. The vehicle must transmit a code to the parcel deliverer, which the deliverer can use to open the boot of the vehicle once.

[9] The case studies described in this chapter – *parking assistance system*, *parcel delivery* and *charging pole search* – are purely exemplary and do not claim to be universally valid for systems of this kind. Depending on the type of product, the technical requirements for the product and the organisation, development can take place differently.

[10] The classification of safety criticality serves here only to illustrate the case study. No conclusions are to be drawn as to what classification a system of this kind actually to be developed would have. This can only be done within the framework of a hazard and risk analysis carried out by functional safety experts in relation to a specific development project.

6. The parcel delivery system must receive and process the code.
7. The vehicle must check the code before opening the boot once and open the boot only after successful verification.

It is the OEM's responsibility to implement items 3, 5 and 7, which will be examined in more detail here. It is assumed that a hazard and risk analysis has not identified any particular safety criticality and therefore ISO 26262 does not need to be complied with. On the other hand, there are a multitude of requirements with regard to cybersecurity. For example, it must be ensured that communication between the vehicle and the parcel deliverer cannot be intercepted and read by unauthorised persons and that the input of the access code is sufficiently protected against hacking attacks. In order to communicate with the parcel deliverer and to open the trunk, existing control units of the vehicle must be upgraded. The requirements of Automotive SPICE must be met. In addition to these boundary conditions, quality management defines subject-specific requirements, such as with regard to the maintainability of the system by workshops or deactivation of the system by the OEM.

Example charging station search

An app shall be developed for drivers to find charging poles and obtain further information about the respective charging poles (supplier, costs, availability, etc.). As this requires neither development, modification nor activation of vehicle ECUs nor implementation nor adaptation of safety-critical functions, Automotive SPICE and ISO 26262 do not need to be taken into account. There are cybersecurity requirements (GPS data of the app user should be transmitted in encrypted form), but these do not exceed the security requirements of similar apps. The quality management itself has no specific requirements for the app, but must ensure the quality of the implemented functional requirements and the maintainability of the app.

The determination of the product type and the determination of the quality requirements in an agile environment is basically no different from the conventional procedure: Functional requirements have to be determined and boundary conditions have to be fulfilled as with waterfall development. However, typically the type of documentation and the transmission of requirements to the development differs. Instead of a conventional requirement description in the form of a requirement specification, the requirements for agile development are included in the product backlog and usually written in the form of user stories. The requirements should meet the INVEST criteria (*independent, negotiable, valuable, estimable, small, testable*) as far as possible (Wake 2003).

Example parcel delivery

Excerpt from the product backlog:

Requirement X: "As an OEM, I can generally deactivate the operation of the parcel delivery service in order to be able to (temporarily) deactivate the service in the event of discovery of security problems that cannot be resolved in a timely manner, numerous complaints or other reasons."

Requirement Y: "As a garage employee, I can update the software of the control unit to communicate with the parcel deliverer in order to keep the control unit up to date."

etc.

The user stories should be supplemented by acceptance criteria and explained and, where necessary, made more specific in a later dialogue with the development team. It is the task of the Product Owner, in consultation with the requester, to break down the business requirements into fine granular business requirements so that the requirements can be implemented within an iteration (*ready for sprint*).

In particular, quality requirements and boundary conditions can already have been formulated in previous projects and integrated into the development at that time. These requirements are usually already specified in detail. Even if such a documented detail specification of particular requirements is unusual in an agile environment, it still makes sense to use such existing documents for new agile developments.

Example parking assistance system

A number of quality requirements have already been recorded in the DOORS requirements management system in previous developments of parking assistance systems. The new development will use JIRA as a tool to manage the product backlog. The new functional requirements are gathered there. A transfer to JIRA of the DOORS requirements known from previous projects has proved impracticable, so that a backlog entry in JIRA with a brief description and a link to the corresponding DOORS entry is created for each business requirement in DOORS. In order to ensure the bidirectional traceability required in Automotive SPICE, the IDs of the backlog entries are also maintained in DOORS.

In agile development, requirements can be introduced into the development process even after implementation has begun. This also applies to quality requirements. It is crucial that quality management is informed about new planned functional requirements and that the resulting quality requirements can be added to the backlog by the Product Owner.

Example charging station search

The app shall be expanded to include functions for customisation (e.g., charging station search near the stored apartment and workplace addresses). This results in legal boundary conditions regarding data

protection, such as compliance with the GSPR in Europe. These boundary conditions must also be taken into account when the development is carried out.

4.1.2. Decision for or against agility

Simple or complex project?

Agility is not an end in itself but serves the effective implementation of complex development projects. In general, the following question should be asked before any development takes place: Does it make sense to develop a system in an agile way? The crucial argument for agility is uncertainty regarding business requirements and technical implementation of complex products. Whether this complexity is present in the development of a system cannot be answered in general terms. For example, the complexity of control units in vehicles is likely to continue to increase, making agile methods more useful. Ultimately, this can only be decided individually for each system to be developed. The requirements initially recorded, including those relating to the quality of the product, are an important decision-making criteria here. If it can be assumed that these requirements are described completely, correctly and with sufficient precision, it would be more appropriate to develop according to a classical approach.

Low or high change costs?

In addition, high change costs can impede or even prevent agility. It should be clarified whether it is possible, for example, to develop embedded systems consisting of hardware and software in an agile way – at least from a purely functional point of view, that is, without consideration of boundary conditions by laws and standards. The prerequisite for an economical, agile approach is low costs for necessary revisions of the increments implemented to date in order to implement further requirements. However, these costs seem to be very high for hardware: New features may require revision of the hardware design and thus creation of new production tools. At the very latest, the idea of continuously rolling out new versions of embedded systems, combined with recall actions after each development iteration, shows that agility that includes continuous delivery as with pure software development would be economically unreasonable. If, however, one only considers pure development and excludes production and delivery, it becomes clear that in many cases hardware can be developed economically in an agile way. To this end, only the term "deliverable" needs to be expanded: If "deliverable" in pure software development means delivery to the customer, "deliverable" here should be interpreted as "ready for volume production." Production itself then takes place in a conventional phase-oriented manner and according to well-defined process steps.

Example parking assistance system

Let's assume that for the new parking assistance system, the hardware of sensors needs to be revised. Frequent adaptation of the production facilities for the production of the sensors would in this case involve unreasonably high costs. One solution might be that the Product Owner decides to give priority to product backlog items relating to the sensors or to the extended functional range of the sensors. The entire development, including the hardware development of the sensors, would then be carried out in an agile manner in an interdisciplinary team. Only when the hardware of the sensors has sufficiently matured through prototype development will the sensors be manufactured as usual. Further software development using the new sensors continues to be agile.

Boundary conditions compatible with agility?

Legal regulations, industry-specific guidelines and in-house processes can apply to the development of products and must be respected. If a product is to be developed in an agile manner, it is necessary to check whether the agile procedure is in principle compatible with these regulations, guidelines and internal instructions – hereinafter referred to as *standards* – and to what extent the agile procedure may have to be modified. To this end, the following process is proposed, which should be carried out once in a company when introducing agile methods and, if necessary, again when changes are made to relevant standards.

1. **Identify conflicts between standard and agility**: For each standard, it must be checked whether agile development is to be brought into line with the standard. In many cases it may not be possible to make a general statement about the entire standard. Rather, process groups, process steps, process results, artefacts or other *rules* – hereinafter summarised under the term rule – must be examined with regard to compatibility with the agile approach.

2. **Technical evaluation of rules**: If a rule contradicts agile principles, practices or the intended agile framework, it should be checked whether the rule makes sense technically. Partial or complete loss of agility due to compliance with the rule must be taken into account. This is the hardest part of the assessment: for each rule there should be reasons for maintaining the rule. The big challenge is to weigh whether these reasons justify a waiver or restriction of agility. If a rule is judged to be so important that loss of agility should be accepted, this rule should remain unchanged.

3. **Change rules in line with the standard if necessary**: If the retention of a rule does not outweigh the loss of agility, it should be checked whether the rule can be adapted by tailoring in such a way that the rule conforms to agile development.

4. **Negotiate rules if necessary**: If reinterpretation or tailoring is not possible within reason, the suspension of the rule or its amendment may be negotiated. In particular, the rules of standards that have been defined in-house come into question for this.

Section 3 describes the standards Automotive SPICE, ISO 26262, AUTOSAR and the maturity level assurance for new parts according to VDA with regard to compatibility with agile development.

Example parking assistance system

As already described above, AUTOSAR, Automotive SPICE and the requirements for ASIL D in accordance with ISO 26262 must be taken into account in the development of the parking assistance system assumed in this case. The compatibility of these standards with agile development is explained in section 3. It is assumed here that an agile approach is possible in principle. The degree of independence I3 required by ISO 26262 for confirmation measures for ASIL D is ensured by the fact that the auditors work closely and continuously with the development team on behalf of the Product Owner but belong to different departments and thus report to different superiors than the members of the agile team. In addition, the company generally demands that the degree of maturity of all new parts be tested in accordance with the VDA standard. Due to the poor compatibility of this standard with agile development (see section 3.4), the following changes are agreed for the development of the parking assistance system:

- For maturity level assessement RG 1 and RG 2, an initial product backlog must be created instead of a complete requirement specification, in which all known boundary conditions and other requirements, which are already comprehensively documented as requirement specification items, are referenced in the product backlog.
- Maturity level assessment RG 3 is not required.
- For maturity level assessment RG 4, only the requirements regarding continuous integration, including automated regression tests, must be fulfilled.
- Maturity level assessment RG 5 only covers implementation of the minimum requirements.

Example parcel delivery

For parcel delivery, measures to secure cybersecurity in particular must be implemented. It is expected that ISO Standard 21434 will have to be taken into account in future (ISO/SAE 2017). As this is still under preparation and the Common Criteria make process guidelines too rigid (CC 2009), the company decides that at least one cybersecurity expert must be part of the development team. The treatment of the VDA standard for maturity level assurance for new parts is the same as for the parking assistance system.

Example charging station search

Neither industry standards need to be taken into account for the charging station search, nor does the development have to be taken into account for the entire vehicle development, so that their degree of maturity need not be tested according to VDA.

4.1.3. Organisation

As with conventional development projects, the basic organisation of the development project must be clarified in advance. In an agile environment, the following questions, among others, must be answered:

- Which agile framework should be used for the development?
- If applicable, what does the scaling model look like?
- Who takes on the role of Product Owner, Agile Team Master or Scrum Master? To which organisation or organisational unit do these roles belong?
- How should the development team be composed? To which organisations/organisational units should their members belong?
- How is the transparency of the product backlog, development status and other artifacts ensured? Who should be involved in the flow of information?
- Who can/should participate in the agile rituals?

Depending on the development initiative, the question can be answered very differently.

Example parking assistance system

Overall responsibility for the development of the parking assistance system remains with the technical development of an OEM. The sales department provides the Product Owner. A supplier should develop the software for the system. In addition, individual suppliers of system components (e.g., for sensors) are integrated partly via the OEM and partly via the Tier 1 supplier. The development at the supplier is carried out according to the Extreme Programming Framework. The supplier has set up three agile teams that are loosely coupled and essentially coordinate their activities according to the SAFe approach (see section 1.9.2). Each of these teams has its own agile team master and its own team Product Owner. On the OEM side, there is a Quality Management department which, in coordination with the Product Owner of the sales department, will incorporate requirements and formally approve the entire system developed before each delivery (see section 4.4).

Example parcel delivery

The development of the parcel delivery system for cars is being carried out by a consortium consisting of an OEM, a parcel delivery company and an online retailer. Each of these members provides a team. The teams are closely linked and coordinate according to LeSS (see section 1.9.1). The technical development of the OEM provides the Product Owner. The OEM's team develops according to Scrum. In the OEM's team, a cybersecurity expert actively participates in the development. Here, too, the OEM's *quality*

management department accompanies the development and can obtain information on the current development status at any time.

Example charging station search

A single team is to develop the charging station search app on behalf of an OEM. The development is to take place first according to Scrum, later according to Kanban if necessary. The OEM provides the Product Owner; the development team and the ScrumMaster are provided by the contractor. The OEM's quality management department communicates with the Product Owner throughout the entire development process, sets requirements and participates in backlog definitions and sprint reviews as required.

4.2. Evaluate suppliers

4.2.1. Supplier audit

Many companies in the automotive industry require a supplier audit to be carried out before the order is placed. Depending on the type of product to be developed, different standards must be adhered to. For agile development, it must be checked whether and to what extent these standards are compatible with an agile approach (see also sections 3 and 4.1.2).

In contrast to supplier audits in conventional development projects, the following criteria should be taken into account for agile development:

- The supplier should demonstrate expertise in development according to agile values and principles and, depending on the agile framework, appropriate qualifications or certificates for the agile roles assumed by them.
- The supplier should demonstrate expertise, tools and methods for quality-assured Continuous Integration and Continuous Delivery.

4.2.2. Contract arrangement and trial period for suppliers

The formulation of contracts with suppliers is not the task of quality management. Nevertheless, the type of contract plays a major role in quality management, since the contract should include requirements for the evaluation of suppliers. In contrast to the classic waterfall model, agile development does not have a complete specification that could serve as the basis for a contract for work and services. Instead, alternatives to the conventional fixed-price agreement with binding requirement specification must be found (see section 1.5). For example, the principal could be

given the option of terminating the contract and assigning another supplier. Evaluation of the supplier then takes place during the collaberation and less in advance through a supplier audit.

Example charging station search

Two companies have been short-listed to implement the charging station search app. The principal agrees a service agreement with both companies in which the companies undertake to develop increments that can be delivered every two weeks. Four months after the start of development, the principal evaluates the performance of the two companies, chooses one of them for further cooperation and terminates the contract with the other.

In order to ensure conformity with the company's internal guidelines on temporary employment, the contracts stipulate that the official contact person of the contractor is always present during the sprint planning phase and authorised to accept orders.

Example parcel delivery

In-house guidelines stipulate that contracts of this kind may only be awarded as contracts for work and services. In order not to prevent agility through such a contract, however, only the target definition of the development and general business requirements can be the contractual basis. In order to ensure the character of the contract as a contract for work and services, the Scrum procedure with a sprint length of three weeks is contractually agreed and an acceptance of each increment by the Product Owner within the framework of the sprint review is prescribed; in the event of unsatisfactory implementation of the sprint target, the contractor is liable. Both sides are granted the right to terminate the contract at the end of a sprint with a defined period of notice. The Definition-of-Done stipulates that at a sprint end the source code must be made available to the client and the current system must then be sufficiently documented.

4.3. Assure maturity level

The maturity level assurance for software in general is described in detail in section 2. Accordingly, a fundamental distinction must be made between quality management by the client and quality management by the contractor (see Section 2.2). While the client's quality management validates quality from the outside, the contractor's quality management supports quality verification as part of the agile development team.

4.3.1. Verification of software quality by the development team

Section 2.3 describes in detail how a development team can test the quality of software. In an agile development team, it is the task of the entire team to ensure quality. In many cases, quality assurance experts can and should be part of the development team, regardless of whether they belong to the same organisational unit as the programmers or not. These quality assurance experts carry out a large part of the quality assurance tasks within the team and are not solely responsible for quality but only as part of the development team.

Example parking assistance system

Due to the high quality requirements for the parking assistance system, especially with regard to functional safety, the agile teams each have several quality assurance specialists with a focus on test automation, performance and functional safety who are part of the development team. These quality assurance experts in particular identify safety-relevant test cases, support the developers in implementing the corresponding unit tests, verify the integrated software and configure the DevOps tools so that all critical tests are automatically tested as regression tests before a new version is delivered. The quality assurance experts belong to a "Quality Assurance" organisational unit that is independent of the development department.

Example charging station search

Since the app for the search for charging stations mainly has high requirements for the usability of the app but is not critical with regard to cybersecurity and functional safety, the development team consists exclusively of software developers, some of whom have expertise in the areas of DevOps and user experience.

4.3.2. Validation of the business requirements by the client

External assurance of software quality, that is, without direct involvement in development as part of the team, is described in detail in section 2.5. In essence, it is the client's task to validate the implementation of the functional requirements, boundary conditions and quality requirements in purely business terms, while at the same time leaving it to the development team to decide how to implement these requirements technically.

How such quality management can be organised in practice depends strongly on the type of product and the associated requirements and cannot be defined in general terms. In the following, possible validation approaches are outlined exemplarily and in excerpts.

Example parking assistance system

When it comes to assuring the degree of maturity of the parking assistance system, the focus is on the quality criteria of functional suitability, particularly functional safety, as well as time behaviour and reliability. These quality criteria can also be validated relatively easily from the outside. The integrated system, whether in a virtual test bench or the physical component or physical vehicle, is tested according to these criteria. The testing itself does not differ fundamentally from the testing of a conventionally developed system, but these tests are carried out more frequently and not only at the end of product development. In addition, quality management on this level focuses on explorative tests; mass testing of various test cases is carried out within the team and, if possible, on the unit level (see also section 2.3.3).

Example parcel delivery

The focus of maturity assurance is on the topic of cybersecurity in addition to user usability and the functional capability of the entire service. Quality management commissions security specialists who check the security measures on the basis of penetration tests. The results of these tests are incorporated into further development.

Example charging station search

The charging station search app is uncritical with regard to functional security and vulnerability to hackers. On the other hand, the requirements for maintainability, usability and portability are high. Precisely because the app is to be brought onto the market quickly and then successively expanded to include further functions, a high degree of modularity, reusability, analysability, modifiability and testability must be guaranteed. To this end, quality management regularly records the speed at which functional requirements are implemented. Slowing down of the speed is an indication for a lack of maintainability of the software.

Quality management also focuses on the usability of the app and continuously coordinates this with sales and the Product Owner. Quality management conducts usability tests with potential users of the app and discusses the results with the agile team. The Product Owner records the resulting requirements in the backlog and prioritises them. In addition, quality management initiates performance and load tests by the supplier and evaluates the results. These tests also do not even take place at the end of the project but are carried out several times during the development period.

4.3.3. Automation of quality assurance

Due to the short iterations, test automation is of great importance in agile development. It is a prerequisite for enabling continuous integration and continuous delivery, if required, at a reasonable cost (see section 2.3.2). Depending on the type of product, quality management should demand this from the supplier. For quality management, this also means that bugs or other

shortcomings discovered during or after development should be covered by an automated regression test in the future, if possible, and do not have to be repeated manually.

Example parking assistance system

The various development teams of the suppliers operate their own continuous integration systems for their components to be developed. By means of these systems, the unit-tested software units are imported into the respective integration environments and automatically tested there on the component level.

In addition, the OEM as the client provides the development teams with a platform on which the developed hardware can be installed and to which the software can be uploaded online. Changes are not applied to this acceptance instance until the components have successfully passed all tests at the respective supplier. The acceptance instance should nevertheless reflect the current state of development as far as possible. The acceptance instance is used by quality assurance experts of the agile development teams as well as by the Product Owner, quality managers and quality experts of the customer. The OEM reports the defects there, which are mostly detected exploratatively. If possible, test functions derived from this will be implemented on the supplier's unit level and automatically executed as regression tests in the supplier's system in future.

4.3.4. Continuity and cooperation

The conventional waterfall model provides phases for quality assurance. There are no such phases in agile development. Quality assurance accompanies both the entire development process within the development team and the client's maturity test. For quality management on the client side, this means that it should be informed about implemented product backlog items (PBIs) during an iteration, be able to participate in reviews as required and in consultation with the Product Owner, and be familiar with the release plan of the Product Owner so that it can test the product more extensively before a planned release. The aim is to provide the agile team with feedback on the PBIs and increments implemented as early as possible.

Such continuity in the cooperation between customer and supplier presupposes a different type of cooperation than with a classic approach. Quality management – whether represented alone by the Product Owner or by one or more stakeholders from a quality management department – participates as personally as possible in agile events such as backlog refinement or review. It knows the developers and is available to the development team as a partner.

Example parking assistance system

In particular, the OEM's quality management checks the implementation of its own subject-specific quality requirements and functional safety requirements. Requirements for eliminating bugs and deficits are

only included and prioritised in the product backlog in agreement with the Product Owner. Quality managers, safety engineers and other quality experts of the OEM take part in discussions on requirements gathering and analysis as well as in the presentation of the increments and are thus in dialogue with the agile teams.

Example parcel delivery

The quality management department knows the current release plan for the parcel delivery system and mainly assumes coordinating tasks before delivery by confirming that the Product Owner or security expert has carried out the necessary security tests. When implementing its own quality requirements, such as with regard to maintainability by garages, the department is also involved in the development process via the Product Owner, as are other stakeholders, that is, it participates in sprint reviews as required and can also impose new requirements during development.

4.4. Release products

Depending on the type of product, quality management may have to formally release a delivery of the product to the customer or a transfer to the manufacturing process. This approval process is neither part of the agile manifest nor one of the usual agile frameworks. Instead, the Product Owner, for example, can decide independently according to Scrum whether an increment should be delivered or not.

Nevertheless, a formal release process by quality management does not represent a fundamental contradiction to an agile approach as long as quality management has tested the product during the previous development period and reported defects to the team promptly, so that final acceptance should not result in extensive change requirements and release can take place without delay. It should be absolutely avoided that quality managers or other stakeholders report deficits that have been contained in the product for a long time and that could have been identified at an earlier point before the planned release or even impose new requirements and thus prevent a timely release. This is avoided by all stakeholders working closely and continuously with the Product Owner and the agile team throughout the entire development period, focusing on what is necessary from a business and legal point of view.

Depending on the product class and context, release cycles in an agile environment can be much shorter than in a conventional development process. Particularly for short cycles, the release must be able to take place quickly. A high degree of test automation, which at least partially eliminates the need for time-consuming manual testing, makes this possible.

Example parking assistance system

The development of the parking assistance system is embedded in the overall vehicle development with a specified date for the Start-of-Production (SOP). Although the parking assistance system is developed incrementally in short iterations of a few weeks, there is only one release date. The required formal acceptance and release by the quality management takes place as usual, but due to the preceding cooperation between the agile teams and the quality management and due to the high level of test automation that is now possible, it can take place faster than is usual with a conventional procedure. The effort for quality management is not concentrated shortly before the SOP but is spread out over the entire development period.

Example charging station search

The app for charging station search should be rolled out as quickly as possible in order to make it possible to evaluate usage behaviour and acceptance in the market in a timely manner. The first release is only a Minimum Viable Product (MVP), which will be functionally extended successively. The Product Owner decides when the product is a MVP and should be rolled out as the first release. Updates are then performed approximately every two to three sprints; the Product Owner also decides on the rollout date in this case. Quality management, which is in contact with the Product Owner throughout the entire development period and is informed about the development status, has made sure that there is a comprehensive suite of regression tests that can be viewed at any time by quality management and that the processes for continuous integration and continuous delivery are implemented. For this reason, quality management releases the respective increments promptly without extensive checks.

4.5. Observe the field and derive lessons learned

Field observation is basically carried out in the same way for products that have been developed in an agile manner as for products that have been conventionally developed. The failure correction processes established in the automotive industry can also be used for agilely developed products.

If a product is changed by updates, insights from the field can be used in an agile way in the current development. In particular, the agile team can check assumptions with the help of A/B tests. Frequent releases allow lessons learned to be derived promptly. These can be used by the Product Owner and stakeholders to identify and set new requirements, as well as by the agile team to optimise the product technically.

Example parking assistance system

After the SOP, the new parking assistance system will be used in the field for the vehicle project for which the system was developed. Complaints about the parking assistance system are handled by the OEM as usual via the failure correction process and, if necessary, lessons learned are derived for the development of future comparable systems.

Example charging station search

The Product Owner analyses the usage behavior by evaluating the databases and log files with the help of analysis tools and by analysing the relevant KPIs. From this, they can conclude what value the single PBIs implemented so far have and what value the PBIs not yet implemented will probably have. Accordingly, they prioritise the product backlog and add new entries to it.

4.6. Living agile principles

In the previous sections, the various tasks of quality management were examined from the perspective of agile product development. The agile values as formulated in the Agile Manifesto are crucial for successful agile development. These values apply to all agile development and should be taken into account in all activities, including quality management, and practised by all stakeholders. The four agile values and their significance for quality management are therefore summarised below.

4.6.1. Individuals and interactions over processes and tools

The quality of the product depends decisively on the competence of all employees involved and their collaboration. This is especially true for agile development and also applies to the quality assurance experts within the team and the quality managers, who introduce and validate quality requirements on the client side. In an agile environment, personal interaction of experts and joint finding of solutions often replaces comprehensive process guidelines. This demands a lot from the employees with regard to their professional, communication and team competence. It changes the role and nature of the work of quality management: it is less the quality police that verifies formal compliance with standards but rather the partner of the agile team that supports development by taking the customer's point of view and contributing to and ensuring development. Quality managers should think and act in a result-oriented and less process-oriented way. These competencies – the ability to work in a team, communication skills and a focus on results – should be nurtured among quality management employees and should be reflected in a corresponding talent

management system. Quality managers who demand compliance with formal standards without questioning their meaningfulness in the given context are unsuitable for agile teamwork.

4.6.2. Working software over comprehensive documentation

For the customer it is crucial that the product works and has the required quality. Documentation of the product and product development is not an end in itself but should serve this purpose (see also section 2.6). Conventional quality management assesses the development status primarily on the basis of documents and demands what is in them accordingly. However, quality management should understand and accept that an agile team may communicate and archive information with each other and with its environment in a different way than is common in the classical context. The team may achieve the purpose of documents in a different way. It is crucial that the business requirements are implemented. Quality management should focus on this and examine the usefulness of documentation in this respect.

Intensive testing of the product to be delivered is a prerequisite for ensuring the required functionality and quality of the product. This applies to agile as well as to conventional development. Due to the short iterations and frequent revisions of the product, the test scope is particularly large in agile environment. Test automation plays a key role here. Only a high degree of test automation can ensure that agile development with frequent deliveries and limited resources for manual testing does not become fragile development. The task of quality management is to set appropriate requirements with regard to quality-assured continuous integration and continuous delivery and to support and ensure their implementation.

4.6.3. Customer collaboration over contract negotiation

The Agile Manifesto emphasises the importance of close cooperation between the customer and the supplier. Cooperation based on trust is more important than the contractual arrangement between the two parties. For the client's quality management this means that it continuously accompanies the development process instead of checking adherence to contractual commitments in a quality assurance phase on the basis of the documents alone. Personal contact and interaction with the agile team – whether with the Product Owner for clarifying requirements, the Agility Master team for procedural questions or the development team for questions during a development iteration – is an important prerequisite for an agilely developed, quality-assured and high-quality product. Quality management should focus on continuous monitoring and participation in agile events, in particular backlog definitions and sprint reviews.

4.6.4. Responding to change over following a plan

Agile development offers the customer the opportunity to formulate requirements even during product development and to incorporate them into the development process. This also applies to quality requirements. Quality deficits that are detected during development and perhaps only after delivery can be remedied by new quality requirements in ongoing development. Critical requirements, especially with regard to functional safety or cybersecurity, should certainly be known and implemented before delivery. Uncritical quality requirements, for example with regard to usability, can be implemented by the agile team at a later date, provided that installation of updates does not involve a great deal of effort. With early deliveries and the use of analysis tools, quality management has the opportunity to gain experience in the field and derive new requirements. Quality management thus assumes an important task in closing the agile loop.

4.7. Conclusion

Quality management in the automotive industry encompasses the same fields of activity in an agile environment as in a conventional environment. Even for agile development, quality requirements must be introduced, suppliers evaluated, maturity levels inspected, deliveries released, the field observed and lessons learned taken into account for further development. It is also important to consider and adhere to binding quality standards.

However, the way these activities are organised and carried out may differ significantly from the quality management of waterfall-type development. Accordingly, agile teams and quality managers must interpret standards differently than usual and at the same time understand and achieve the goals associated with the standards. This requires both a deep understanding of the individual standards and their meaningful application to the respective development project and, if necessary, adaptation of agile practices.

If an agile approach for a development project makes sense and is possible, the quality should be managed and safeguared according to the agile principles:

- Personal interaction of quality managers and quality assurance experts with the team and its members is more valuable than an assessment according to what is filed.
- Continuous guidance over the entire product lifecycle is more effective than isolated phases of quality assurance.
- Validation of the quality requirements and their adaptation and honing even during the development process is more effective than pre-specification of all potentially required quality characteristics in advance.

- Iterative adaptation of the methods and tools of quality management to the respective project is better than the use of a general canon of methods for all developments.

APPENDIX

A. Quality characteristics of software products according to ISO/IEC 25010

"Functional suitability

This characteristic represents the degree to which a product or system provides functions that meet stated and implied needs when used under specified conditions. This characteristic is composed of the following subcharacteristics:

- **Functional completeness.** Degree to which the set of functions covers all the specified tasks and user objectives.
- **Functional correctness.** Degree to which a product or system provides the correct results with the needed degree of precision.
- **Functional appropriateness.** Degree to which the functions facilitate the accomplishment of specified tasks and objectives.

Performance efficiency

This characteristic represents the performance relative to the amount of resources used under stated conditions. This characteristic is composed of the following subcharacteristics:

- **Time behaviour.** Degree to which the response and processing times and throughput rates of a product or system, when performing its functions, meet requirements.
- **Resource utilization.** Degree to which the amounts and types of resources used by a product or system, when performing its functions, meet requirements.
- **Capacity.** Degree to which the maximum limits of a product or system parameter meet requirements.

Compatibility

Degree to which a product, system or component can exchange information with other products, systems or components, and/or perform its required functions, while sharing the same hardware or software environment. This characteristic is composed of the following subcharacteristics:

- **Co-existence.** Degree to which a product can perform its required functions efficiently while sharing a common environment and resources with other products, without detrimental impact on any other product.
- **Interoperability.** Degree to which two or more systems, products or components can exchange information and use the information that has been exchanged.

Usability

Degree to which a product or system can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use. This characteristic is composed of the following subcharacteristics:

- **Appropriateness recognizability.** Degree to which users can recognize whether a product or system is appropriate for their needs.
- **Learnability.** Degree to which a product or system can be used by specified users to achieve specified goals of learning to use the product or system with effectiveness, efficiency, freedom from risk and satisfaction in a specified context of use.
- **Operability.** Degree to which a product or system has attributes that make it easy to operate and control.
- **User error protection.** Degree to which a system protects users against making errors.
- **User interface aesthetics.** Degree to which a user interface enables pleasing and satisfying interaction for the user.
- **Accessibility.** Degree to which a product or system can be used by people with the widest range of characteristics and capabilities to achieve a specified goal in a specified context of use.

Reliability

Degree to which a system, product or component performs specified functions under specified conditions for a specified period of time. This characteristic is composed of the following subcharacteristics:

- **Maturity.** Degree to which a system, product or component meets needs for reliability under normal operation.
- **Availability**. Degree to which a system, product or component is operational and accessible when required for use.
- **Fault tolerance.** Degree to which a system, product or component operates as intended despite the presence of hardware or software faults.
- **Recoverability.** Degree to which, in the event of an interruption or a failure, a product or system can recover the data directly affected and re-establish the desired state of the system.

Security

Degree to which a product or system protects information and data so that persons or other products or systems have the degree of data access appropriate to their types and levels of authorization. This characteristic is composed of the following subcharacteristics:

- **Confidentiality.** Degree to which a product or system ensures that data are accessible only to those authorized to have access.
- **Integrity.** Degree to which a system, product or component prevents unauthorized access to, or modification of, computer programs or data.
- **Non-repudiation.** Degree to which actions or events can be proven to have taken place, so that the events or actions cannot be repudiated later.
- **Accountability.** Degree to which the actions of an entity can be traced uniquely to the entity.
- **Authenticity.** Degree to which the identity of a subject or resource can be proved to be the one claimed.

Maintainability

This characteristic represents the degree of effectiveness and efficiency with which a product or system can be modified to improve it, correct it or adapt it to changes in environment, and in requirements. This characteristic is composed of the following subcharacteristics:

- **Modularity.** Degree to which a system or computer program is composed of discrete components such that a change to one component has minimal impact on other components.

- **Reusability.** Degree to which an asset can be used in more than one system, or in building other assets.
- **Analysability.** Degree of effectiveness and efficiency with which it is possible to assess the impact on a product or system of an intended change to one or more of its parts, or to diagnose a product for deficiencies or causes of failures, or to identify parts to be modified.
- **Modifiability.** Degree to which a product or system can be effectively and efficiently modified without introducing defects or degrading existing product quality.
- **Testability.** Degree of effectiveness and efficiency with which test criteria can be established for a system, product or component and tests can be performed to determine whether those criteria have been met.

Portability

Degree of effectiveness and efficiency with which a system, product or component can be transferred from one hardware, software or other operational or usage environment to another. This characteristic is composed of the following subcharacteristics:

- **Adaptability.** Degree to which a product or system can effectively and efficiently be adapted for different or evolving hardware, software or other operational or usage environments.
- **Installability.** Degree of effectiveness and efficiency with which a product or system can be successfully installed and/or uninstalled in a specified environment.
- **Replaceability.** Degree to which a product can replace another specified software product for the same purpose in the same environment."

(ISO/IEC 2010)

B. Base scenario for compatibility of ISO 26262 and agility

Description

Basically, there are a variety of possibilities for combining agile development and the requirements of ISO 26262. The following presents a possible scenario that addresses the challenges presented in section 3.2.2 and is presented in Figure 12. The scenario is designed to implement the entire safety lifecycle of ISO 26262 in an agile setting. For such an agile setting, however, an initial hazard and risk analysis should be carried out at the beginning as the basis for further development. This can be done in an initial concept phase. Each iteration of agile development then covers the entire security lifecycle. However, the phases prescribed there do not have to be carried out in the planned fixed sequence. It is only necessary to ensure that the necessary measures are applied and found in the corresponding artefacts. Therefore, the results of the concept phase must also be questioned and possibly revised in each iteration, so that a new assessment of the criticality of an increment with regard to functional safety is definitely possible. To enable this implementation, the following organisational scenario is presented.

The basic idea of the base scenario is to leave the full responsibility for the product in the agile team and nevertheless meet the required roles of ISO 26262 and, in particular, enable all degrees of independence in the confirmation measures so that the complete security lifecycle is covered by the scenario.

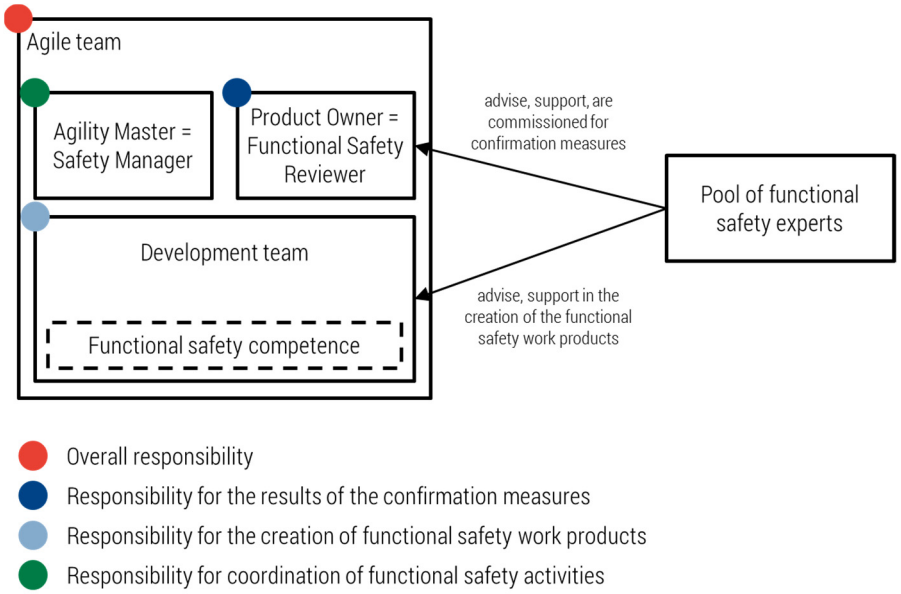

Figure 12: Structural representation of the base scenario

The development team is responsible for creating the safety work products demanded by ISO 26262. The development team must therefore have the necessary expertise. It is quite conceivable

that the team will call in support from functional safety experts. However, basic competence must be present in the development team so that it takes full responsibility for creating the work products.

The role of the safety manager, which in this scenario is assumed by the Agility Master, verifies that the required safety-specific work products are created and that the development team adheres to the safety plan. By integrating the safety manager role into the agile team, the agile principle of self-responsibility is fulfilled. The organisational responsibility of the safety manager can be easily combined with the role of the Agility Master. The Agility Master is responsible for the organisation of the development team and can therefore – provided they have a basic understanding of ISO 26262 and competence with regard to functional safety – assume the additional organisational effort of acting as a safety manager. Another important role named in ISO 26262 is the project manager. Their responsibility can easily be supplemented by the already mentioned role of the safety manager. Therefore, in the scenario shown, both roles are combined in the person of the Agility Master. This integration of the roles is explicitly permitted according to ISO26262 (ISO/FDIS 26262-2:2018, 6.4.2.4 – NOTE 1).

While the Agility Master is responsible for general execution of the security activities, the Product Owner is assigned responsibility for the confirmation measures. There is no explicit role description in ISO 26262 that matches the responsibilities required by this approach. However, there are some roles that can be consolidated in this new role of functional safety reviewer, such as functional safety assessor. Through assignment to this and related roles, the functional safety reviewer gains direct responsibility not only for implementation of the confirmation measures, but also for their results. However, the Product Owner does not necessarily have to perform these new tasks themselves but can ask for support from the functional safety expert pool. According to ISO 26262, this division of execution tasks, with full responsibility remaining with the role at the same time, is explicitly permitted and can be found in the description of the previously mentioned role of the functional safety assessor, for example. The functional safety reviewer may belong to an organisational unit that differs from all members of the development team, thus fulfilling the requirement of their independence.

Evaluation of the scenario in relation to the challenges

The scenario presented offers an agile solution to the challenges described in section 3.2.2, which are briefly discussed here.

Competence: In the basic scenario, all competencies relating to functional safety are represented. Safety-specific work products are created by the development team, with the confirmation measures carried out by the Product Owner. Advice can also be obtained from external experts.

Responsibility/independency: In the scenario, the roles that assume responsibility for functional safety according to ISO 26262 are integrated with appropriate independence, so that overall responsibility for the developed product can be assumed by the agile team. In accordance with ISO

26262, the role of the safety manager is not subject to any requirements regarding independence from the development team. As an alternative to the basic scenario, the safety manager can therefore also be located within the development team. This can be a reasonable decision in order to avoid burdening the role of Agility Master with tasks that require additional, considerable technical responsibility or decision-making authority and thus necessary functional safety competence, as would be required by the role of safety manager.

Phase-oriented safety lifecycle / changes: The basic scenario is designed to allow the execution of all activities and the creation of all work products of the safety lifecycle in each iteration. The phase orientation is overcome in this way. This also means that changing requirements during development is not a problem. The agile team itself is responsible for continuous adaptation of all affected safety-specific work products and has the necessary expertise.

Iteration length / team size: The challenge remains that the effort required to perform or adapt all necessary activities of the safety lifecycle requires longer iterations. In addition, many of the required competencies are distributed among small agile teams, which can be difficult in practice even with a generous build-up of competencies in the area of functional safety in the company.

Conclusion

The basic scenario presented shows how the challenges can be met by integrating competence and responsibilities from the area of functional safety into the agile team. In this way the required self-organisation of the development team can be maintained and, in principle, the entire safety lifecycle can be executed in agile iterations of an agile team. Due to the limited team size and iteration length, the scenario is primarily suitable for smaller projects. In the automotive industry, however, projects or products often have a size that cannot be handled by a single agile team. Therefore, from a practical point of view, application in large projects requires scaling of the agility, for example by splitting the development into several agile teams that are loosely or closely coupled (see section 1.9).

In addition, it is necessary to ensure appropriate granularity of the development scopes for which the individual agile teams are responsible. Even the development of a system (within the meaning of the common definition of an automotive OEM) can be a task that is too complex for a single agile team. This means that the development of systems must be broken down into functions or even requirements accordingly. With such hierarchical structuring of the development scope, accompanying structuring of agile development teams would be conceivable. Such an implementation of agility, however, opens up new challenges regarding integration of iteration results on the system level. However, these can be considered independently of the challenges of ISO 26262 and are already treated within the framework of approaches such as “Scaled Agile”. Finally, the compatibility of agile principles and the special requirements of ISO 26262 can be confirmed in principle. Of course, the practicability of the exact form and implementation of the agile approaches must be evaluated beforehand in every project context.

C. Bibliography

A-SPICE (2017): Automotive SPICE Process Assessment / Reference Model. Version 3.1. *VDA QMC Working Group 13 / Automotive SIG*. 2017.

Bass, Len / Weber, Ingo / Zhu, Liming (2015): *DevOps: A Software Architect's Perspective*.

Beck, Kent / Beedle, Mike / Bennekum, Arie van / Cockburn, Alistair / Cunningham, Ward / Fowler, Martin / Grenning, James / Highsmith, Jim / Hunt, Andrew / Jeffries, Ron / Kern, Jon / Marick, Brian / Martin, Robert C. / Mellor, Steve / Schwaber, Ken / Sutherland, Jeff / Thomas, Dave (2001a): *Manifesto for Agile Software Development*. http://agilemanifesto.org

Beck, Kent / Beedle, Mike / Bennekum, Arie van / Cockburn, Alistair / Cunningham, Ward / Fowler, Martin / Grenning, James / Highsmith, Jim / Hunt, Andrew / Jeffries, Ron / Kern, Jon / Marick, Brian / Martin, Robert C. / Mellor, Steve / Schwaber, Ken / Sutherland, Jeff / Thomas, Dave (2001b): *Principles behind the Agile Manifesto.* http://agilemanifesto.org/principles.html

Besemer, Frank / Karasch, Timo / Metz, Pierre / Pfeiffer, Joachim (2014): *Clarifying Myths with Process Maturity Models vs. Agile*. White Paper.

Bondar, Arina / Steckinger, Otmar / Dussa-Zieger, Klaudia (2015): "Wie sag ich's meinem Assessor?" *HANSER automotive 11-12 / 2015*, Carl Hanser Verlag, Munich.

CC (2009): *Common Criteria for Information Technology Security Evaluation*. ISO 15408.

Crispin, Lisa / Gregory, Janet (2009): *Agile Testing – A Practical Guide for Testers and Agile Teams*. Addison-Wesley.

Deming, W. Edwards (1986): *Out of the Crisis*. The Mit Press.

Diebold, Philipp / Zehler, Thomas / Richter, Dominik (2017): "How Do Agile Practices Support Automotive SPICE Compliance?" *ICSSP'17*, July 2017, Paris, France.

Foegen, Malte / Kaczmarek, Christian (2016): *Organisation in einer digitalen Zeit*. Second edition.

ISO/FDIS (2018): *ISO/DIS 26262 Road vehicles – Functional safety*. ISO/FDIS Final Draft International Standard, 2018-07.

ISO/IEC (2010): *ISO/IEC 25010 System and software quality models*. ISO/IEC.

ISO/SAE (2017): *ISO/SAE AWI CD Road Vehicles – Cybersecurity engineering* (under development), announced at https://www.iso.org/standard/70918.html, 2018.

Johnson, Jim (2002): *ROI, It's Your Job*. Third International Conference on Extreme Programming, Alghero, Italy.

Larman, Craig / Vodde, Bas: Introduction to LeSS (2018). https://less.works/less/framework/introduction.html

McConnell, Steve (2006): *Software Estimation. Demystifying the Black Art*. Redmond, Washington.

Pink, Daniel H. (2009): *Drive – The surprising truth about what motivates us*. Riverhead Books, New York.

Rupp, Chris / die SOPHISTen (2009): *Requirements-Engineering und -Management*. 5th edition. Carl Hanser Verlag Munich Vienna.

Safety Research & Strategies, Inc. (2013): *Toyota Unintended Acceleration and the Big Bowl of "Spaghetti" Code*. http://www.safetyresearch.net/blog/articles/toyota-unintended-acceleration-and-big-bowl-%E2%80%9Cspaghetti%E2%80%9D-code

Schwaber, Ken (2013): *UnSAFe at any speed*. https://kenschwaber.wordpress.com/2013/08/06/unsafe-at-any-speed/

Schwaber, Ken / Beedle, Mike (2002): *Agile Software Development with Scrum*. Prentice Hall.

Schwaber, Ken / Sutherland, Jeff (2017): *The Scrum Guide – The Definitive Guide to Scrum: The Rules of the Game.*

Stacey, Ralph D. (2000): *Strategic Management and Organisational Dynamics: the challenge of complexity*.

Thomas, Dave (2015): *Agile is Dead*. GOTO Conference 2015.

VDA (2009): *Product creation – Maturity Level Assurance for New Parts*. VDA-Blaugold-Band, 2nd revised edition, October 2009.

VDA (2017): *Automotive SPICE® – Guidelines*. 1st edition, September 2017.

Wake, Bill (2003): *INVEST in Good Stories, and SMART Tasks*. https://xp123.com/articles/invest-in-good-stories-and-smart-tasks/

Wells, Don (1999): *Pair Programming*. http://www.extremeprogramming.org/rules/pair.html

West, Dave (2011): *Water-Scrum-Fall Is The Reality Of Agile For Most Organizations Today*.