An Overview of Quality Assurance Practices in Agile Methodologies

Olli P. Timperi

***Abstract*—The focus of literature and debates of agile methodologies has been on the development activities while quality assurance practices of different agile methodologies have received less attention and an overall picture is missing. This paper collects quality assurance practices of different agile methodologies together and analyzes them. Accuracy of quality assessment, costs, information to be gathered, timing, empirical evidence, and concrete guidance are analyzed from each quality assurance practice. Based on the findings, quality assurance is analyzed at a methodology level from a good enough quality viewpoint. The findings show that agile methodologies propose a wide range of quality assurance practices that cover the most important areas. The biggest problem found was that testing was not described in appropriate detail. The results indicate that most agile methodologies have focused on validation at the expense of verification. Thus, most of the studied methodologies lack the balance in quality assurance and are not capable of producing good enough quality software. In the current situation, it is recommended to combine quality assurance practices of different methodologies in order to get good enough software delivered to the customer.**

***Index Terms*—agile methodologies, quality assurance, software quality, software verification and validation.**

1. Introduction

SEVERAL agile methodologies have been developed in the last few years. Agile methodologies are software development methodologies that are adaptive and collaboration-oriented. The focus of literature and debates of agile methodologies has been on the development activities while quality assurance practices of different agile methodologies have received less attention and an overall picture is missing. However, quality must be addressed if these methodologies are to be applied to practice in the software industry. This paper collects quality assurance practices of different agile methodologies together and analyzes them. The methodologies are analyzed from a good enough quality (Bach, 1997) perspective. Good enough quality is a framework for analyzing the quality of a software product in terms of its readiness to delivery. The aim of agile methodologies is to deliver business value rapidly by delivering working software frequently. Therefore, the good enough quality concept is suitable for analyzing how agile methodologies have used quality assurance practices to gain confidence that the implemented software is of good enough quality before deliveries. Based on the results, strengths, weaknesses, and applicability of the methodologies from the quality assurance viewpoint are discussed and improvement areas as well as future research areas are suggested.

The research problem can be stated in the following two questions. What quality assurance practices are proposed in different agile methodologies? How does the combination of these practices within a methodology support achieving quality from a good enough quality perspective?

The main objective of the research is to collect quality assurance practices from different agile methodologies to a single paper and analyze them to gain deeper knowledge about the state of quality assurance in these methodologies. The analysis of quality assurance practices focuses on the accuracy of product quality assessment, cost of quality assurance, information that is gathered to support decision making, and timing of all these elements. In addition, empirical evidence and concrete guidance are addressed. Each methodology should provide these pieces of information in order to be applicable to practice. The second objective of the research is to analyze and discuss how the combination of proposed quality assurance practices aims to achieve good enough quality at a methodology level. The sub objective of the research is to suggest improvement areas in quality assurance for different agile methodologies.

The research studies six agile methodologies: adaptive software development, Crystal Clear, dynamic systems development method, extreme programming, feature-driven development, and Scrum. Quality assurance is only analyzed from the product quality viewpoint. Process quality and its improvement are not part of this study because process improvement is generally driven by organizational policies instead of a software development methodology. This limitation of scope also justifies the use of the good enough quality concept in analyzing quality assurance.

The research was conducted by doing a literature study. The literature study focused on books that describe the methodologies. Articles from AbiInform, ACM, CiteSeer, CSA, IEEE, Link, and ScienceDirect databases were also used for finding empirical evidence of quality assurance practices.

The results of a literature study are generally not directly applicable to practice. However, the purpose of this study is not to show that a certain methodology would be superior in practice, rather the aim is to present and analyze if quality assurance is addressed in a methodology well enough to be applied to practice as such or if other quality assurance practices should be combined with the ones proposed by the methodology. Because of the purpose of this paper, a literature study is a justified method for conducting the research.

Most related studies of agile methodologies have focused

either on agile methodologies from other than the quality assurance perspective (e.g. Abrahamsson et al., 2003) or on certain quality assurance practices used in some agile methodologies individually (e.g. Williams et al., 2000). The lack of studies from the quality assurance perspective is mostly due to the fact that agile methodologies are relatively new and they have not yet been widely used in the software industry. This paper attempts to present an overview of quality assurance in agile methodologies which is currently missing.

The paper is structured as follows. First, the methodologies are introduced and described briefly. Second, the analytical framework and its application are explained. The next chapter answers the first research question and provides an analysis of quality assurance practices suggested by different agile methodologies. Next, based on the findings, the methodologies are analyzed at a methodology level from the good enough quality perspective. Strengths, weaknesses, and applicability are also discussed from the quality assurance viewpoint. Finally, the results of the paper are summarized and future research topics are suggested.

2. METHODOLOGIES

This study focuses on six agile methodologies: adaptive software development (ASD), Crystal Clear, dynamic systems development method (DSDM), extreme programming (XP), feature-driven development (FDD), and Scrum. The selection of these methodologies is based on a study (Abrahamsson et al., 2003) that analyzed which agile methodologies provide described processes for different life cycle phases. If at least the requirements phase and some testing life-cycle phases were covered, the methodologies were selected to this study in order to make sure that the studied methodologies have enough quality assurance related practices. Crystal family (Cockburn, 2001) consists of several methodologies and only Crystal Clear was chosen because it is intended for relatively similar sized projects as the other methodologies in this study. The methodologies are described briefly in the following.

2.1 Adaptive Software Development

ASD (Highsmith, 2000) is based on the complex adaptive systems theory. ASD is intended for high-speed and high-change projects that are developed by self-organizing teams. The development is iterative and incremental. The iterations contain three overlapping phases: speculate, collaborate, and learn. The emphasis is on enabling emergent behavior which requires simple rules and rich connections between people. Many of the proposed development practices are collaboration-oriented and encourage learning. Time-boxed iterations are used to obtain frequent results and to force engineering trade-offs. Quality is mainly handled by planning software in joint application development sessions and reviewing the implemented software at the end of iterations.

2.2 Crystal Clear

Crystal Clear (Cockburn, 2001; Cockburn, 2002) is a part of Crystal family which is a set of methodologies developed for different situations. Each methodology in the family has a color that represents how much coordination and quality assurance guidance is provided. Crystal Clear, the methodology chosen to this study is aimed for small teams developing software. It is iterative and incremental in nature. It defines a set of practices that a process must have to be called Crystal Clear but the exact practices are not strictly defined and it is possible for example to use practices from other methodologies in Crystal Clear. The emphasis is on collaboration and on process tuning by reflection. Frequent reviewing of software with customers and an on-site customer are the main quality assurance practices in Crystal Clear.

2.3 Dynamic Systems Development Method

DSDM (Stapleton, 1997) is developed by the DSDM consortium in the UK and it originates from rapid application development. It is intended for building business systems rapidly with fixed time and resources. DSDM relies heavily on prototyping in most development activities. Prototyping is used to elicit functional requirements and to develop working software. DSDM proposes a pragmatic view to quality; the emphasis is on early validation while technical quality can be sacrificed. The cornerstones of DSDM are time-boxing and frequent deliveries. Iterations are time-boxed and contain several checkpoints and reviews that force a usable delivery at the end of iterations.

2.4 Extreme Programming

XP (Beck, 2000; Jeffries et al., 2001) is a combination of engineering practices that support each other when used together. XP proposes 12 engineering practices and these practices provide the stability needed in high-change projects. XP is based on short iterations and incremental development with constant feedback from both the customer and other developers. Most of the practices of XP are aimed at quality assurance and in particular getting timely feedback. XP provides concrete guidance to its practices and an off-the-shelf solution. This distinct feature separates it from other agile methodologies that are described at a more general level.

2.5 Feature-Driven Development

FDD (Palmer & Felsing, 2002) is a modeling oriented methodology. FDD is based on several best practices and it emphasizes design and building activities. The requirements are captured first by constructing a domain object model and using it as a basis for requirements elicitation. Development projects start with requirements gathering and planning phases which are followed by iterative and incremental development. The development team is divided into feature teams led by an experienced chief programmer and assisted by class owners who are less experienced programmers. Design and implementation phases are separated and the results of both phases are inspected. Inspections are the main quality assurance practice but testing is also mandatory.

2.6 Scrum

Scrum (Schwaber & Beedle, 2001) is an agile methodology

that focuses on the management side of software development. It is based on empirical management and it does not state engineering practices. Therefore, it can be superimposed on top of other agile methodologies that provide engineering guidelines. Scrum process itself consists of development in iterations called sprints. The requirements are captured in prioritized order in a product backlog and in a sprint backlog for the current sprint. Sprint planning and review practices are described for managing software projects. Daily management is handled by scrum meetings in which participants answer three questions regarding what they have done since the last scrum meeting, what they will do between now and the next scrum meeting, and what problems they have.

3. Analytical Framework

This paper focuses on quality assurance practices in agile methodologies. A practice is defined as a recommended approach, employed to prescribe a disciplined, uniform approach to the software life cycle (IEEE, 1996). On the other hand, quality assurance is defined here as a planned and systematic pattern of all actions necessary to provide adequate confidence that the product optimally fulfils customers' expectations, i.e. that it is problem-free and well able to perform the task it was designed for. Therefore, a quality assurance practice is a practice aimed for quality assurance as defined above. Practices that match this definition were searched from the literature.

Quality is considered as a business value in several agile methodologies and the goal is that the software is not perfect but of reasonably good quality. Some agile methodologies use good enough quality (Bach, 1997) explicitly as their quality target. For a piece of software to be of good enough quality means that it has sufficient benefits, it has no critical problems, the benefits sufficiently outweigh the problems, and in the present situation, and all things considered, further improvement would be more harmful than helpful (Bach, 1997). Basically this means that the piece of software is ready for a delivery to the customer. Agile methodologies emphasize delivering working software to customers frequently. Thus, the good enough quality concept is suitable for analyzing how agile methodologies have used quality assurance practices to gain confidence that the implemented software is of good enough quality before deliveries. The properties of good enough quality are used for analyzing each methodology.

Good enough testing (Bach, 1998) is closely related to good enough quality but its focus is on the assessment of a certain quality assurance practice, testing. The good enough testing concept focuses on the analysis of the accuracy of quality assessment, costs of testing, information that is gained through testing and used for decision making, and the timing of these. These characteristics can be generally analyzed from most quality assurance practices with the exception of the accuracy of quality assessment which can be analyzed only from quality assurance practices that specifically assess quality. Because the concept is generally applicable and its context is suitable, the characteristics of the concept are used in the analysis of quality assurance practices. In addition, it is important to know if the quality assurance practices have been applied to practice and is there empirical evidence that they have been successful. To be applicable in practice and to gain consistent results, the usage of the quality assurance practices should be instructed concretely. The characteristics of good enough testing as well as empirical evidence and concrete guidance are analyzed from each quality assurance practice.

The accuracy of quality assessment contains coverage of the assessment as well as its formality. The cost of using a quality assurance practice will be analyzed either as time per iteration or as compared to using alternative practices. Information that is extracted from the application of a quality assurance practice is analyzed. This can be information that is used either to make decisions about the readiness of a piece of software or as an input into other activities. Timing will be explained either related to activities or generally within an iteration, e.g. at the beginning of the iteration. If empirical evidence exists on a quality assurance practice, it will be mentioned and considered as an advantage in terms of quality assurance capability. Concrete guidance means that instructions are given on how to apply the practice, how much it should be applied, and by whom it should be applied. If these pieces of information are given, they will be mentioned.

These elements are used to create an analytical framework for this paper. The framework was chosen because the good enough concepts are very close to the approach that many agile methodologies propose to quality assurance. Therefore, this framework can be used to analyze if agile methodologies are capable to bring the results they claim from the quality assurance viewpoint. First, the quality assurance practices are analyzed separately and then the methodologies are analyzed from the good enough quality viewpoint.

4. Agile Quality Assurance Practices

Based on the literature study, this section presents what quality assurance practices were recognized and answers the first research question: What quality assurance practices are proposed in different agile methodologies? The findings are presented in the following with a classification by the purposes of the practices. First, requirements gathering and validation practices are analyzed. That is followed by verification practices and finally practices for achieving internal quality are analyzed. Summaries of the findings are represented in tables. Each table presents quality assurance practices, summaries of analyzing the characteristics stated in the analytical framework chapter, and methodologies that propose them.

4.1 Requirements Gathering and Validation Practices

Requirements gathering and validation practices answer what should be done and has the right thing been done. These are closely connected with customer collaboration which can be seen as a validation practice. The following requirements gathering and validation practices were found during the literature study. The findings are summarized in Table I.

4.1.1 Demonstration of Software

Demonstration of software means that software implemented during the iteration is demonstrated by executing the software to customers and management who validate the results. ASD, Crystal Clear, DSDM, and Scrum suggest using demonstration of software through executing business scenarios. In ASD these demonstrations are called customer focus group reviews but the idea is the same as in the other methodologies. Demonstration of software is done at a review point which is at the end of iterations in ASD and Scrum. In DSDM and Crystal Clear, demonstrations may be held several times during the iteration. All of these methodologies give concrete guidance on how these demonstrations should be conducted and who should be present as well as a few practical tips. The results of these demonstrations are a list of change requests and accepted parts of a system. This information is used for planning the next iteration. The demonstration sessions last for approximately a day per demonstration. The assessment of the quality of the software is rather informal and covers basic business scenarios. Therefore acceptance testing is required to get an adequate level of confidence of the quality of the system. No evidence exists that demonstration of software in review sessions would be better than other validation techniques.

4.1.2 Joint Application Development

Joint application development (JAD) sessions are structured, facilitated workshops that bring together cross-functional people in order to produce high-quality deliverables in a short period of time (Highsmith, 2000). JAD sessions can be used to produce many deliverables including requirements and prototypes. DSDM proposes using JAD sessions for initial prototyping and requirements gathering. ASD suggests using JAD sessions for eliciting and outlining requirements as well as for setting the project's mission. FDD proposes using JAD sessions to develop an overall domain model and elicit requirements from it. JAD sessions are used at the beginning of iterations. They last usually one day or less and they may be repeated until the goals have been achieved. Studies have shown that JAD sessions are a cost effective and fast technique to develop requirements (Carmel et al., 1992). Concrete guidance is given on how to prepare and conduct a JAD session including roles and responsibilities. Evidence exists (Carmel et al., 1992) that using JAD sessions reduces requirements and design defects.

4.1.3 Joint Planning Meeting

A joint planning meeting is a requirements gathering practice used in Crystal Clear, XP, and Scrum. In a joint planning meeting, customers and developers come together to discuss requirements, ask questions, and confirm that people understand the requirements in a similar way. Joint planning meetings are conducted at the beginning of iterations. Joint planning meetings last only a day or less except for the planning meetings of the very first iteration which can take longer. The cost of using joint planning meetings is relatively cheap compared to other requirements gathering practices but the requirements are gathered on a relatively high level. Thus, all the methodologies that suggest joint planning meetings suggest also using an on-site customer to achieve the necessary elaboration of high level requirements. Concrete guidance is given on roles and responsibilities of the participants of joint planning meetings. No empirical evidence exists that using joint planning meetings result in better requirements and quality than by using other requirements gathering practices.

4.1.4 On-Site Customer

On-site customer is a practice where a customer's representative is available for the developers at the development site full-time. The customer's representative has to know what the system should do and the developers should ask questions concerning requirements when they are not sure what the system should do exactly. On-site customer is proposed by XP, Crystal Clear, and Scrum. In addition, DSDM proposes continuous user involvement which means together with prototyping that there must be an on-site customer. Cost of using an on-site customer is high especially in small projects and this makes its use hard to justify. Part-time user involvement might be more appropriate from the

TABLE I
REQUIREMENTS GATHERING AND VALIDATION PRACTICES

Practice	Coverage and Formality	Cost	Information	Timing	Concrete Guidance	Empirical Evidence	Methodologies
Demonstration of software	Business scenarios, informal	One day per demonstration	Change requests and data for planning next iteration	At the end of iterations or in the middle at checkpoints	Yes	No	ASD, DSDM, Scrum, Crystal Clear
Joint application development	Functionality with some detail, facilitated	Few days per iteration	Requirements	At the beginning of iterations	Yes	Yes	ASD, DSDM, FDD
Joint planning meeting	Functionality on a high level, informal	One day per iteration	Requirements	At the beginning of iterations	Yes	No	XP, Scrum, Crystal Clear
On-site customer	Ambiguous requirements, informal	Full-time customer	Confirmation of ambiguous requirements	Full-time	Not meaningful	No	XP, Scrum, DSDM, Crystal Clear
Prototyping	Most of the functionality, informal	Cheaper than traditional methods	Requirements	All the time during development	No	Yes	DSDM

cost perspective. Guidance is not given on how much developers should use the on-site customer but this is an almost impossible practice to be concretely instructed. Thus, the lack of concrete guidance is not a problem with this practice. No empirical evidence exists that using an on-site customer would improve quality but studies have shown that the distance between people influences how often they communicate. One can deduce that an on-site customer should therefore improve communication and informal validation.

4.1.5 Prototyping

Prototyping is a practice that is used to create quickly a look-alike of the actual system with limited functionality and capabilities. DSDM uses prototyping as its requirements gathering method as well as development method. In DSDM prototypes are not thrown away, but instead evolved into the actual system. The development of other agile methodologies can be seen also as evolutionary prototyping, but prototyping is considered here as an explicitly defined practice rather than feedback from incremental deliveries. Using prototyping is cheaper than spending time on requirements specification in a traditional way (Boehm et al., 1984). Prototyping is used throughout the development but concrete guidance is not given on how prototyping should be done. Empirical evidence exists in several studies (Gorden & Bieman, 1995) that prototyping can be used to produce quality software with few defects.

4.2 Verification Practices

Verification practices are intended to check that a given artifact conforms to its specifications. The verification practices that were found are summarized in Table II and are analyzed next.

4.2.1 Automated Acceptance Testing

Automatic acceptance testing means that all acceptance tests should be automated rather than manually executed. Acceptance tests are defined by customers as in traditional acceptance testing. XP and DSDM propose using automated acceptance testing. Automation must be gained by using tools, for example record and replay tools for user interface (DSDM) and tools for executing the system against an application programming interface (XP). Tests in DSDM should not be scripted, but rather recorded at the user interface level when users execute the system. On the other hand, XP relies on scripted tests that should preferably not be executed at the user interface but at the application programming interface. Tests should cover everything that should work like normal acceptance testing. Costs of using automated testing depend on the amount of retesting and regression testing compared to changing existing tests. DSDM's approach of using non-scripted tests that are recorded at user interface is a cost efficient technique because tests are fast to modify and run. XP's style of using scripted tests makes updating tests costly but running the tests cheap compared to manual testing. Passed tests are used as a measure of project progress in XP. DSDM uses automated tests mainly to make sure that nothing has broken, i.e. for regression testing. Automated acceptance tests should be ready by the middle of iterations in XP and run daily after that. DSDM uses checkpoints within iterations for timing acceptance tests. No evidence exists that automated acceptance tests were superior to manual testing or vice versa. The benefit of automation in agile methodologies comes from the fact that software is changed frequently and it is likely that retesting and regression testing should be done often. Therefore, using automated tests may save both time and money.

4.2.2 Daily Builds with Testing

Daily builds with smoke and regression testing is a practice that has been used in the software industry for a long time and can be considered as a best practice (Cusumano & Selby, 1997). Daily builds refer here to building the whole software system frequently, at least daily but preferably several times a day. When the build is completed, it is tested by running

TABLE II
VERIFICATION PRACTICES

Practice	Coverage and Formality	Cost	Information	Timing	Concrete Guidance	Empirical Evidence	Methodologies
Automated acceptance testing	Everything that should work, automated	Cheap to run, costly to create and update	Project progress measure, regression testing	Tests should be ready by the middle of iterations and run at least daily afterwards	Yes	No	DSDM, XP
Daily builds with testing	Varies	Usually not significant but depends on the speed of build and tests	Find integration and regression defects	Every day or several times a day	Varies	No	FDD, Scrum, XP, Crystal Clear
General testing	Unknown	Unknown	Unknown	During iterations, detailed timing left to developers	No	No	ASD, DSDM, FDD, Scrum, Crystal Clear
Inspections	Representative sections, formal	Good defect removal-cost ratio	Find defects and propose changes	After inspected artifact has been completed	Yes	Yes	ASD, FDD
Pair programming	All programming, informal	Similar to programming alone	Knowledge transfer, defect prevention	All the time during development	Yes	Yes	XP
Test-driven development	Everything that could break, automated	Similar to testing afterwards	Find defects immediately, integration readiness	All the time during development	Yes	Yes	XP

usually automated tests that check that the core of the system works as previously and new features work together with the old ones. Crystal Clear, FDD, Scrum, and XP propose that daily builds should be used and the builds should be tested. Scrum and FDD do not take a stand on how thorough testing should be done but XP requires that all unit tests should be run. Crystal Clear suggests automated regression testing to be used. Costs of using daily builds depend on the time spent on building and running tests. To decrease the costs XP suggests making tests fast or even optimizing their execution time (Jeffries et. al). However, the cost of using daily builds with testing is likely to pay its costs back by giving feedback early. Some anecdotal evidence (Karlsson et al., 2000) has been presented but the effect of using daily builds versus not using them is left to common sense, although one can guess that using daily builds is a good way for finding integration defects.

4.2.3 General Testing

General testing refers to many agile methodologies' approach to testing. They propose that testing should be done throughout the development but the details are not stated. General testing covers all types and levels of testing. Because neither concrete guidance, level of coverage and formality, nor information to be gathered are given, costs and efficiency of this kind of testing depend completely on the development team. Neither can evidence be presented for this kind of unclearly described practice. ASD, Crystal Clear, DSDM (except for acceptance testing), FDD, and Scrum have this approach to testing. They simply state that testing must be done and suggest doing it regularly throughout the iterations, but that is all. Crystal Clear proposes using automated tests that should be run frequently but no concrete guidance is given. FDD goes as far as suggesting that most organizations have already a working system testing process and it can be used as such with FDD (Felsing & Palmer, 2002). However, no proof for the claim is given. This leaves a major area of verification open to questions and speculations with the methodologies that suggest only testing at a general level.

4.2.4 Inspections

An inspection is a formal evaluation technique in which software requirements, design, or code are examined in detail by a person or group other than the author to detect faults, violations of development standards, and other problems. The participants must prepare for the inspection by inspecting an artifact beforehand according to inspection guidelines. ASD and FDD suggest using inspections. FDD proposes both design and code inspections while ASD suggests only code inspections. Inspections are used for achieving also internal quality of the source code as well as promoting knowledge transfer and learning. FDD and ASD propose using checklists that determine which kinds of defects inspectors are looking for. The content of these checklists determine how detailed and comprehensive assessment of the inspected artifact is performed. FDD suggests using inspections for representative sections of each programmer's source code and only if several serious defects are found, the other sections of the programmer's code are inspected. This reduces costs of using inspections while keeping the comprehensiveness of quality assessment high. ASD does not take a stand on what should be inspected. Several studies have shown that inspections are a cost-effective practice to find defects (Fagan, 1976; Jones, 1996). In addition, some defects found in inspections are different from the defects found with testing (Jones, 1996). Both FDD and ASD suggest doing code inspections directly after the implementation rather than waiting to the end of iterations. Both ASD and FDD give concrete guidance on how to do inspections. In FDD inspections are done within a feature team while in ASD the participants of inspections are not as clearly stated. The results of an inspection in both methodologies are changes suggested to the inspected artifact and a decision if the artifact is accepted or if it should be reinspected after the found defects are corrected.

4.2.5 Pair Programming

Pair programming is a practice where programming is done by two people sitting at a single computer. The other person is called a driver and his responsibility is to write the code and to focus on the current matters at hand. The other person is called a partner or a navigator and his responsibility is to check that the code written by the driver is correct and to think ahead. XP suggests using pair programming for all production code all the time. The cost of doing this is relatively equal to the traditional way of programming alone according to some studies (Williams et al., 2000). In addition, using pair programming has reduced the amount of defects in several studies (Williams et al., 2000; Jensen, 2003). However, there is a huge difference in the reduction of defects between these studies which makes their reliability questionable. Pair programming is suggested by XP to replace inspections but few studies (Müller, 2003) have been made that would support this idea and more studies are needed to justify the substitution. Pair programming is not used to create information for decision making but instead reducing defects and transferring knowledge. Concrete guidance is given on pairs' roles and rotation of pairs which is used for achieving knowledge transfer within the development team.

4.2.6 Test-Driven Development

Test-driven development is a practice in which unit tests are written before the source code and ran directly after the implementation is complete. Test-driven development forces the source code to be testable and guarantees that unit tests are written. XP proposes using test-driven development for all production code. XP suggests writing tests for everything that could break which is a very high coverage. Because tests are written before source code, the resulting code is different than if tests were written after the implementation and there is evidence that test-driven development reduces defects (George & Williams, 2003; Williams et al., 2003). The cost of doing test-driven development is relatively similar to writing tests afterwards (George & Williams, 2003; Williams et al., 2003). The information from the unit test results is used to determine whether the implemented code is good enough to be integrated

which requires that the implemented code passes all unit tests. XP provides concrete guidance on what should be tested. Tests should be automated which requires tool support.

4.3 Practices for Achieving Internal Quality

In addition to validating and verifying that software functions as it should, the internal quality of the software must be taken care of. Internal quality means here mainly maintainability of software. The practices found are summarized in Table III and are analyzed next. Some of the verification practices are also used for the same purpose but the following practices focus only on internal quality.

4.3.1 Coding Standard

Coding standard is a set of rules that developers must adhere to and it states how everyone is expected to format the source code. It contains for example naming and indentation rules. This practice improves maintainability because everyone is familiar with the style of source code. Coding standard is used in XP, Crystal Clear, DSDM, and FDD. The latter two combine using static analysis tools with this practice to automate checking that everyone follows the coding standard. Introducing the use of a coding standard increases development costs temporarily but as developers get used to it, the coding standard decreases costs of changing other developers' code, e.g. during maintenance. Information as such is not gathered by using a coding standard but existing code can be evaluated against a coding standard and parts that should be fixed may be found out by using e.g. a code analyzing tool. Crystal Clear, DSDM, FDD, and XP provide guidance on how to use a coding standard by saying that a language specific coding standard should be used and state the main contents that the coding standard should have. Evidence exists (Fang, 2001) that using a coding standard improves code quality and makes maintenance of software easier and cheaper.

4.3.2 Collective Code Ownership

Collective code ownership is a practice where anyone can change any piece of code anytime. XP uses this practice to its source code. Costs of using collective ownership come from potential conflicting changes but otherwise it is similar to using personal code ownership. By using collective code ownership developers can do changes faster and learn from what other developers have done. XP guides using a version control system and active communication among developers while applying this practice. No evidence exists that collective code ownership would be superior compared to personal code ownership but some studies (Nordberg, 2003) discuss different situations where either one might be a better choice. Crystal Clear has taken a similar approach and mandates using a code ownership model which may be collective, personal, or some other code ownership model.

4.3.3 Personal Code Ownership

Personal code ownership means that the person who writes a piece of code is responsible for changing and developing the piece of software in the future as well. This practice is used to keep the internal quality of the source code at a good level because it is believed that responsibility makes it more motivating to keep code quality high. Both Scrum and FDD suggest using this practice and it should be used always when programming. Negative cost impacts of using this practice may occur if changes are postponed when a person responsible for a piece of software is busy. The benefit of using personal code ownership is that impacts of doing changes are understood well and negative impacts can be avoided. Both Scrum and FDD give simple guidance regarding personal code ownership; the person who writes the code is responsible for updating it in the future. As mentioned, there is no evidence that personal code ownership would be better than the other code ownership models (Nordberg, 2003).

4.3.4 Refactoring

Refactoring means changing the source code without changing its observable behavior. XP uses refactoring to keep internal quality of source code good when code is changed frequently to avoid degrading code quality. In practice, the other methodologies may also need to use refactoring but that is not stated explicitly. Any part of source code can be refactored anytime when a developer sees an opportunity in XP. However, XP gives concrete guidance on how and when to do refactoring which helps to make only important and helpful refactorings. After refactoring, all the unit tests should be run and passed like after every change in XP. The cost of refactoring depends on how often refactorings are done and how much they improve the source code. If only necessary refactorings are made, the cost of using refactoring is similar to development without them because the time spent in

TABLE III
PRACTICES FOR ACHIEVING INTERNAL QUALITY

Practice	Coverage and Formality	Cost	Information	Timing	Concrete Guidance	Empirical Evidence	Methodologies
Coding Standard	Full code coverage, formal (DSDM, FDD), informal (XP, Crystal Clear)	Initial adaptation	Parts of code that need to be changed	All the time	Yes	Yes	DSDM, FDD, XP, Crystal Clear
Collective code ownership	Not applicable	Potential conflicting changes	Knowledge transfer	All the time	Yes	No	XP
Personal code ownership	Not applicable	Potential waiting before changes implemented	Deep understanding of impacts of changes	All the time	Yes	No	FDD, Scrum
Refactoring	As needed, informal	Potential unwanted changes	Keeping design simple	As needed	Yes	Yes	XP

refactoring is saved later with better maintainability. No information is gathered during refactoring but it is used to keep the design simple and the source code readable. Evidence exists (Kataoka et al., 2002) that refactoring helps improving maintainability of source code.

5. Methodology Level Quality Assurance

In this section, the agile methodologies are analyzed from the good enough quality viewpoint and the implications of the analysis are discussed. This analysis and discussion answers the second research question: How does the combination of these practices within a methodology support achieving quality from the good enough quality perspective. The methodologies are analyzed and discussed separately and the results are summarized in the next chapter. The discussion focuses on strengths, weaknesses, and applicability of the methodologies from the quality assurance viewpoint. In addition, ideas are presented on what areas should be improved within the methodologies to achieve better overall quality assurance from the good enough quality perspective. Table IV presents which quality assurance practices the methodologies suggest to give an overview of the methodologies' quality assurance.

5.1 Adaptive Software Development

ASD uses joint application development to make sure that developers implement features that provide benefits to the customer. At the end of iterations a demonstration of software is arranged to make a check that the results actually provide the benefits that were agreed during joint application development sessions. ASD uses mainly code inspections to find out critical problems and enforce internal quality. Testing is left mandatory but vague because its level and coverage are not instructed. At the demonstration of software meetings, benefits and problems should be known and their effects can be judged. The result of the meeting is a change request list which is used for further improvement. However, if the proposed quality assurance practices have been used thoroughly, the situation at the end of iterations should be such that benefits sufficiently outweigh problems and further improvement would not be beneficial from a business point of view. Thus, at the end of iterations ASD should be capable of providing software that is of good enough quality.

The strengths of ASD from the quality assurance viewpoint are that inspections are an efficient and proven technique to find defects and validation is well handled by combining joint application development with demonstration of software. Together these should be enough to achieve confidence that the piece of software has sufficient benefits and the most critical problems are found. However, ASD does not provide guidance on what, how, and how much should be tested. This is a weakness that must be resolved in an organization using ASD to have confidence that defects that are not found in inspections are found in other ways. Because ASD emphasizes validation rather than verification, from a quality perspective it is not applicable to life-critical systems, but instead business systems that can afford defects in software. To improve ASD testing should be described more concretely or testing from other agile methodologies could be combined with ASD to improve the effectiveness of verification.

5.2 Crystal Clear

Crystal Clear relies on the combination of joint planning meetings and an on-site customer to make sure that requirements are understood correctly. At the end and in the middle of iterations, implemented software is validated by demonstrating the software. If these practices are correctly used, the product should have sufficient benefits to the customer at the end of iterations. Crystal Clear relies on frequent integration and automated regression testing to find out defects. However, the level and coverage of testing is not given and this leaves verification vague. Thus, by using only quality assurance practices of Crystal Clear, the methodology should be capable of guaranteeing benefits but problems might not be known. Therefore at the end of iterations, it is hard to know if the implemented software is of good enough quality.

The strength of Crystal Clear is well-handled validation but the weakness is poorly described verification. As the creator (Cockburn, 2002) of Crystal Clear admits, the methodology is primarily applicable to projects whose quality criteria are loss of discretionary moneys, not stricter quality requirements. Other methodologies of the Crystal family are aimed for such projects. If Crystal Clear was combined with better instructed verification from other methodologies, its weakness could be solved. Especially XP suits well with Crystal Clear because it provides concrete testing guidelines but does not propose practices that would not be considered as being Crystal Clear. In addition, the validation practices of these two methodologies fit well together.

5.3 Dynamic Systems Development Method

DSDM uses mainly extensive prototyping as its requirements gathering practice. Some formality is gained to prototyping by using JAD sessions in initial prototyping. In addition, continuous user involvement and demonstration of software several times during iterations should be enough to make sure that the software does what it is intended to and contains sufficient benefits. Verification and internal quality are not as well handled. Even though coding standard should be used for achieving internal quality, design and maintainability issues are not emphasized. Verification is done through executing the software at the user interface level and retesting the same scenarios automatically. Other testing is not described although the importance of other testing is admitted by the authors of the methodology. Thus, at the end of iterations, benefits are well known and problems should not exist in tested business scenarios. If problems lie elsewhere, they are not known and therefore DSDM is not capable of satisfying the criteria of good enough quality.

The strengths of DSDM are proven requirements gathering and validation techniques. The weaknesses lie in verification which is left to the development organization to decide upon. From a quality perspective DSDM suits best in situations

where the correctness and benefits of software can be seen from the user interface. If DSDM was to be applied to other types of software, verification practices should be taken from other methodologies. However, this might not be a good idea because the methodology is specifically designed for building user interface centric software and it might not be at its best in other situations. DSDM could be improved by instructing testing in more detail and by introducing daily builds.

5.4 Extreme Programming

XP uses joint planning meetings, automated acceptance testing, and an on-site customer to ensure sufficient benefits. On-site customer and joint planning meetings are used to make sure that the developers know what should be done. However, the only practice that is actively used to make sure that benefits exist is using automated acceptance testing that has been instructed by the customer. If the tests cover the functionality well, it should be possible to make sure that sufficient benefits exist in the software. To find critical problems XP proposes test-driven development with automated unit testing, daily builds with full regression testing, pair programming, and automated acceptance testing. In addition, collective code ownership, coding standard, and refactoring are used to guarantee internal quality. These practices should be enough to verify that there are no critical problems in the software. The only doubt comes from the substitution of inspections with pair programming. Currently there is only little evidence (Müller, 2003) that this substitution works which leaves space for speculation. Anyway, XP has a wide range of quality assurance practices whose usage is guided concretely which means that XP should be capable of producing good enough quality software.

The strength of XP is that it has a good balance between the different quality assurance practices and the most important areas are covered. There are no obvious weaknesses but replacing manual validation with automated testing leaves possibly a gap in the validation. XP can be virtually applied to any kind of situation that does not require formal proofing of correctness. Basically this means everything but life-critical systems. XP could be improved by combining it with other agile methodologies that have more validation emphasis while retaining XP's approach to testing and verification. Potential methodologies that could be used with XP in this way are Crystal Clear and Scrum because the fit between the quality assurance practices is excellent.

5.5 Feature-Driven Development

FDD resembles the waterfall style development in its approach towards assessing that benefits exist in the software. JAD sessions are used at the beginning of the project to develop an overall domain model and elicit requirements from it. After that point, customers are involved as necessary. The main purpose of testing and inspections is to verify that the system meets the agreed requirements. Therefore, making sure that the software has sufficient benefits is not handled well. Design and code inspections are used extensively and they are used for assuring internal quality together with coding standard and individual code ownership in addition to assessing functionality. However, FDD does not give guidance on testing but rather it suggests using the process already in place. This may be enough to find the critical problems. But because FDD lacks validation, it does not fulfill the criteria of good enough quality software at the end of iterations.

The strength of FDD is its well instructed use of inspections which are a proven technique. The weaknesses are the lack of validation after initial requirements gathering and vaguely described testing. Because of these factors, FDD is applicable to situations where validation is not as important as verification. To improve validation, FDD could be combined with practices from other methodologies. Perhaps the best fitting candidate would be ASD which has a similar approach to verification but adds feedback and validation in the form of demonstration of software.

5.6 Scrum

Even though Scrum focuses on the management side of software development, it proposes several quality assurance practices that support decision making. Scrum uses joint planning meetings, an on-site customer, and demonstration of software during a review meeting at the end of iterations to validate and to make sure that sufficient benefits exist in the software. Scrum proposes using daily builds with some testing and some sort of testing throughout the iterations. Because the level and comprehensiveness of testing is not specified, these are not enough to get confidence that critical problems do not exist in the software. Because the knowledge of problems is potentially vague, Scrum does not fulfill the criteria of good enough quality.

The strength of Scrum from the quality assurance point of view is that validation

TABLE IV
QUALITY ASSURANCE PRACTICES BY METHODOLOGY

Quality Assurance Practice	ASD	Crystal Clear	DSDM	FDD	Scrum	XP
Demonstration of software	X	X	X		X	
Joint application development	X		X	X		
Joint planning meeting		X			X	X
On-site customer		X	X		X	X
Prototyping			X			
Automated acceptance testing			X			X
Daily builds with testing		X		X	X	X
General testing	X	X	X	X	X	
Inspections	X			X		
Pair programming						X
Test-Driven development						X
Coding Standard		X	X	X		X
Collective code ownership						X
Personal code ownership				X	X	
Refactoring						X

X = the practice is used in the methodology

is handled well. The weakness of Scrum is that it leaves too many things open about verification and testing. Therefore, as the authors of the methodology suggest, Scrum should be combined with another agile methodology. If Scrum is used independently, it is applicable to situations where validation is emphasized and verification is not important. Scrum can be easily combined with other methodologies because it leaves most engineering practices open. It is best combined with XP because the validation practices are easily combined and XP provides verification practices that Scrum lacks.

6. Summary and Conclusions

The aim of this paper was to give an overview of quality assurance in agile methodologies. A wide range of quality assurance practices are proposed by agile methodologies as presented in Tables I-III. The problems are mainly associated with testing in many methodologies because concrete guidance and instructions are not given. In addition, there is no evidence that many of the proposed quality assurance practices have been applied successfully in practice. Timing is well described in most practices which is important in the tight development rhythm of agile methodologies.

Agile methodologies were also analyzed and discussed from the good enough quality perspective. The results indicate that most agile methodologies have focused on validation and making sure that benefits exist in the software at the expense of verification. The exceptions are XP and FDD that emphasize verification over validation. Both approaches lead to problems when it comes to good enough quality that requires balance between the two. Thus, most of the studied methodologies are not capable of producing good enough quality software individually but if quality assurance practices of different agile methodologies were combined suitably, the problems could be solved.

However, none of the methodologies had empirical evidence that the proposed quality assurance practices work together or even that all of the practices worked on their own. Therefore, the methodologies should be studied as whole entities from the quality assurance viewpoint. There is also room for studying the individual quality assurance practices especially in the agile context. Another interesting research topic would be combining quality assurance practices of different agile methodologies and finding out the impacts because the methodologies are presently unbalanced. In the current situation, it is recommended to combine quality assurance practices of different methodologies in order to get good enough software delivered to the customer.

References

Abrahamsson, P., Ronkainen, J., Siponen, M., Warsta, J., "New Directions on Agile Methods: A Comparative Analysis", 25th International Conference on Software Engineering, 2003.

Bach, J., "Good enough quality: Beyond the buzzword", IEEE Computer, 1997, vol. 30, no 8, pp. 96 – 98.

Bach, J., "A framework for Good Enough testing", IEEE Computer, 1998, vol. 31, no 10, pp. 124 – 126.

Beck K., "Extreme Programming Explained: Embrace Change", Addison-Wesley, 2000.

Boehm, B., W., Gray, T., E., Seewaldt, T., "Prototyping versus Specifying: A Multiproject Experiment", IEEE Transactions on Software Engineering, Vol SE-10, No. 3, May 1984.

Carmel, E., George, J.F., Nunamaker, J.F., Jr., "Supporting joint application development (JAD) and electronic meeting systems: moving the CASE concept into new areas of software development", Proceedings of the Twenty-Fifth Hawaii International Conference on System Sciences, 1992, Volume: iii, 7-10 Jan. 1992, pp. 331 -342 vol.3

Cockburn A., "Agile Software Development", Addison-Wesley, 2001.

Cockburn A., "Crystal Clear: A Human-Powered Methodology for Small Teams", http://alistair.cockburn.us/crystal/books/cc/crystalclear.zip, 2002, referenced 25.3.2004.

Cusumano, M. A., Selby, R., W., "How Microsoft builds software", June 1997, Communications of the ACM, Volume 40, Issue 6.

Fagan, M., "Design and Code Inspections to reduce Errors in Program Development", IBM Systems Journl, Vol 15, no. 3, pp.182-211, 1976.

Fang, X., "Using a coding standard to improve program quality", 2001, Proceedings.Second Asia-Pacific Conference on Quality Software, pp. 73-78.

George, B., Williams L., "An Initial Investigation of Test-Driven development in Industry", Proceedings of the ACM symposium on applied computing, March 2003.

Gorden, V.S., Bieman, J.M., "Rapid prototyping: lessons learned", IEEE Software, Volume: 12 Issue: 1, Jan. 1995, pp. 85-95.

IEEE, "IEEE guide for software quality assurance planning", IEEE Std. 730.1-1995, 10 April 1996.

Highsmith J., "Adaptive Software Development: A Collaborative Approach to Managing Complex System", New York, Dorset House Publishing, 2000.

Jeffries R., Anderson, A., Hendrickson, C., "Extreme Programming Installed", Boston: Addison-Wesley, 2001.

Jensen, R. W., "A Pair Programming Experience", CrossTalk, The Journal of Defense Software Engineering, March, 2003.

Jones, C., "Software defect-removal efficiency", IEEE Computer, April 1996, Volume: 29 Issue: 4, pp. 94-95.

Karlsson, E.-A., Andersson, L.-G., Leion, P., "Daily build and feature development in large distributed projects", 2000, Proceedings of the 2000 International Conference on Software engineering, pp. 649-658.

Kataoka, Y., Imai, T., Andou, H., Fukaya, T., "A quantitative evaluation of maintainability enhancement by refactoring", 2002. Proceedings of the International Conference on Software Maintenance, 3-6 Oct. 2002.

Müller, M. M., "Are Reviews an Alternative to Pair Programming?", Conference on Empirical Assessment In Software Engineering, April 2003.

Nordberg, M. E., III, "Managing code ownership", Software, IEEE , Volume: 20 Issue: 2 , March-April 2003, pp. 26-33.

Palmer S. R. and Felsing J. M., "A Practical Guide to Feature-Driven Development", 2002.

Schwaber K, Beedle M., "Agile Software Development With Scrum", Upper Saddle River, NJ: Prentice-Hall, 2002.

Stapleton J., "Dynamic systems development method – The method in practice", Addison-Wesley, 1997.

Williams, L., Cunningham, W., Jeffries, R., Kessler, R., "Strengthening the case for pair programming", IEEE Software, 2000, vol. 17, no 4. pp. 19-25.

Williams, L., Maximilien, M., Vouk, M., "Test-driven development as a defect-reduction practice", 14th International Symposium on Software Reliability Engineering, 2003, 17-20 Nov, pp. 34 - 45.