Product metrics in agile software development

Hanna Kulas

University of Tampere
School of Information Sciences
Computer Science
M.Sc. thesis
Supervisor: Eleni Berki
January 2012

University of Tampere

School of Information Sciences

Computer Science

Hanna Kulas: Product metrics in agile software development

M.Sc. thesis, 52 pages, 15 index and appendix pages

January 2012

The purpose of this study was to discover such a way to use product metrics in agile software development that provides benefits to all affected stakeholders, including both general guidelines and detailed recommendations. The research was based on a literature study, with assessment and aggregation of the identified ideas. It was found that no such previous research exists. The related literature was either more general, considering the rudiments of using software metrics in agile methods, or more focused, concerned with particular metrics or aspects of agile software development. Filling this gap was the motivation for writing this thesis.

The results revealed four areas of measurement to be incorporated in agile software projects: size, quality indicators, defects, and requirements and design. The author recommends specific metrics in each of these areas, including the information on the expected benefits and their receivers, and the means, the best time, and the suitable person to conduct the measurement.

Key words and terms: software metrics, product metrics, measurement, agile software development, agile methods

Contents

1. Introduction

1.1. The history of metrics and the current situation

The history of software metrics is now about 40 years old. It is not clear when exactly this field of science emerged. Ordonez and Haddad [2008] state that software measurement started in the early 1970s, but Fenton and Neil [2000] date it already back to mid-1960s, when the LOC (Lines of Code) metric was introduced as the basis for measuring productivity and effort. The first dedicated book on software metrics, "Software Metrics" by Tom Gilb, was published in 1976. Since that time the area of software measurement has been highly active [Ordonez and Haddad, 2008], and currently there is a large body of research related to software metrics [Kitchenham, 2010].

Despite the academic activity and widespread recognition of their role in software quality, software metrics have not become a significant part of software engineering [Fenton and Neil, 2000] and their utilization in the practice of software development has been marginal and neglected in many organizations [Ordonez and Haddad, 2008; Harjumaa et al., 2008]. According to Fenton and Neil [2000], this is mainly because software metrics failed to address their most significant objective, which is to provide information to support quantitative managerial decision-making during the software lifecycle. Ordonez and Haddad [2008] maintain that the reason is low maturity of the software measurement practice and that much work is still needed to standardize, validate, and incorporate metrics into development practices in the software industry. Kaner and Bond [2004] admit that the immaturity of the field could be one of the reasons, along with high cost of metrics programs and the fact that these programs may do more harm than good due to the use of inappropriate metrics or inappropriate decisions and actions.

1.2. Rationale for research

Publications on software metrics often emphasize the importance of measurement and metrics for software development:

> "Software metrics are useful means in helping software engineers to develop large and complex software systems." [Scotto et al., 2005]
>
> "We can better control software quality and development effort if we can quantify the software." [Li, 1999]
>
> "Managing the agile development process, and any process, requires the collection of suitable metrics (...)" [Berki et al., 2007]
>
> "Metrics and measurement are essential in order to manage the software development work efficiently (...)" [Harjumaa et al., 2008]
>
> "(...) the appropriate use of quality and productivity metrics embodies one of the most important quality management tools any software organization can have." [Bartels et al., 2009]
>
> "The measurement and information fed back to all parties, e.g. developers, managers, customers and the corporation helps in the understanding and control of the software process and products, and the relationships between them." [Basili, 1992]

It can be seen from the above quotations that metrics are considered by the academia to be useful, helpful, vital, and even necessary both for software product and software process. Yet they are not a part of the agile paradigm, and this is the gap that this study is addressing.

Many authors [Bartels et al., 2009; Berki et al., 2007; Georgiadou, 2003; Hartmann and Dymond, 2006; Kunz et al., 2008; Siakas et al., 2005] agree that metrics processes and standards existing in traditional software development (also called: conventional, non-light) cannot be straightforwardly transferred into agile development. According to Hartmann and Dymond [2006], traditional modes of evaluation are incompatible with agile values and principles, and because measurement drives behavior, inappropriate (outmoded) measurement will drive dysfunctional behaviors, such as wasting

resources, distorting team behavior in counter-productive ways, undermining the effort toward culture change inherent in agile work. Kunz et al. [2008] give the following reasons for the need of adaptation of measurement approach to the agile paradigm:

- different product/development lifecycle (iterative), and what follows, different cost of change,
- different practices (especially refactoring),
- inapplicable traditional software product quality standards like ISO/IEC 9126.

Hartmann and Dymond [2006] claim that continuous measurement of both product and process ("whether anecdotal or formalized") is inherent in agile software development. Bartels et al. [2009] express a similar opinion, and specify that this is due to the attributes of agile development such as short iterations, rapid response to change, role planning, and test-driven development. They do not explain why this makes constant measurement of projects, processes, and products necessary, but they add that because working software is the primary measure of progress [Beck et al., 2001], completion of software products must be measured in order to track progress. These claims seem to be in contradiction with the above statements of the very limited use of metrics in the industry. If measurement is really inherent in agile development, then it should not be possible to implement the agile paradigm without it, and yet agile methodologies are currently the most widely used ones across the industry, while metrics are hardly employed. Perhaps the authors mean activities such as following the burn-down charts and constant re-estimation and reprioritization of scope and features. These are some types of metrics, but they concern mostly the development process, not the software product. However, a survey presented by Georgiadou et al. [2003], conducted in software

development organizations in Denmark, Finland, Greece, and the UK, revealed that some measures of the software product are kept by most organizations (about 90% at least), and in many of them (between 47,7% and 67,3%) to a high degree. A potential explanation is offered by Salo et al. [2002] who state that although most software companies have an abundance of data on their processes and products, they often do not analyze it, considering it too time-consuming. Since mere data collection without using it for learning and improvement is meaningless, it cannot be considered proper metrics utilization.

1.3. Motivation

While studies on product metrics for agile software development exist, albeit few, none of them comprehensively addresses the aspects of measurement or suggest a set of metrics with detailed guidelines for their use. There has been no publication so far that would study which product metrics are the most suitable for agile development, how and when they should be used, and what are the benefits for particular stakeholders. This is the gap this thesis is aiming to fill. The expected contribution to the academia is the increased understanding of the specifics of agile processes that affect software measurement (especially product metrics), what is the current state of the research on the topic, and what needs to be done in the future.

Agile methodologies are currently the number one choice in the industry. I have worked in commercial agile projects myself as a software engineer, and I have noticed that the use of metrics is scarce. The study of Ordonez and Haddad [2008] shows that this is not exceptional – to the contrary, it is very common. Considering the numerous benefits software metrics bring, I am interested in using them myself, and for that I need to know an efficient way to do it. Furthermore, much of the research on software metrics is irrelevant to practitioners because of irrelevant scope or content [Fenton and Neil, 2000].

This thesis is then also a much needed contribution to the industry, mainly to project managers and software engineers like myself, who can use the findings in their daily work.

1.4. Research question

The research question of this thesis is:

> *How to use software product metrics in a way that is in compliance with agile values and principles and benefits the affected stakeholders?*

The affected stakeholders are those involved in the development process who can directly benefit from the use of product metrics: project managers, developers, testers, customers, and users.

In order to answer this question, different aspects of measurement must be analyzed, namely: why to measure, what to measure, how to measure, when to measure, who measures and for whom [Berki et al., 2007]. I pursued the answer by reviewing the recent literature on software metrics, agile software development, or both, and assessing and aggregating the ideas found. The references were mostly obtained by searching ACM, IEEE Xplore, Elsevier (ScienceDirect), and SpringerLink databases, some were received from the thesis supervisor, and some were found by using the Google search engine. The keywords used for searching were the following (used in different combinations): software, product, metrics, agile, measurement. I found only several articles directly concerned with the topic, they are all discussed in Chapter 3. I also found tens of articles that were otherwise useful for this research. The list of references includes only the articles used in this thesis.

The outcome of this research are recommendations for the use of product metrics in agile software development, including a set of suggested metrics, and also identification of gaps in the current knowledge and suggestions for future research directions.

1.5. Thesis structure

This thesis is divided into six chapters. Chapter 2 provides background information on software metrics and agile software development, separately. Chapter 3 is a literature review on the use on product metrics together with agile methodologies. Chapter 4 presents the original findings of this research on why, what, how, and when to measure, and who should measure and for whom in agile software development. Chapter 5 contains a discussion of the degree to which the study has answered the research question and what the research has contributed to the literature. Chapter 6 includes a summary, limitations, and recommendations for future work.

2. Background

2.1. Software metrics

Software metrics is a term for a field of software engineering. It is used to describe the wide range of activities concerned with measurement [Fenton and Neil, 2000]. The practice of metrics involves *measures* and *metrics*. Ordonez and Haddad [2008] give good definitions of these terms:

> "A *Measure* is a way to appraise or determine by comparing to a standard or unit of measurement, such as the extent, dimensions, and capacity, as data points. The act or process of measuring is referred to as Measurement."
>
> "(...) a *Metric* is a quantitative measure of the degree to which a component, system, or process posses a given characteristic or an attribute (...)."

In more simple words, a measure is the result of measurement, and a metric is calculated from measures or their combinations [Bundschuh and Dekkers, 2008]. The two terms should not be confused with each other.

Some authors feel there is a need to introduce more terms in order to be more precise:

- A *diagnostic* is a type of metric, but it is temporary, contextual, and local, as opposed to an actual metric, which is long-term, overarching, and organizational. [Hartmann and Dymond, 2006]
- An *indicator* "represents useful information about processes and process improvement activities that result from applying metrics, thus, describing areas of improvement". [Ordonez and Haddad, 2008]

This differentiation, however, is not very common, and if it is present, then the additional terms are sometimes (incorrectly) used interchangeably with "metric". For this study the term "metric" is considered sufficient.

Software *product* metrics focus on measuring attributes of the software product, which is any artifact or document resulting from the software development process [Fenton and Pfleeger, 1998]. Software product entities fall into one of the four categories: specifications, designs, source code, and test data [Ordonez and Haddad, 2008].

Wei Li [1999] simply states that software metrics are measures of software, and adds that their primary use is to help plan and predict software development. According to Fenton and Neil [2000] any software metric is an attempt to measure or predict some attribute of some product, process, or resource – in case of product metrics it is obviously a software product. Attributes can be *internal* or *external*. Internal attributes are those of the system representations, such as design diagrams, source code or documentation [Scotto et al., 2006], and they can be controlled and measured directly, but their metrication is of little value unless there is evidence that the specific metric is related to an external attribute [ISO/IEC, 2001]. Examples of internal attributes of source code are: size, complexity, coupling, modularity, and reuse. External attributes (also called quality properties or quality characteristics [Lincke and Löwe, 2007]) are typically those possessed by the system in execution [Scotto et al., 2006] and cannot be measured directly [Berki et al., 2007; Fenton and Neil, 2000]. They are for example: functionality, reliability, usability, efficiency, maintainability, portability, (these six attributes constitute the quality model of the ISO 9126 standard [ISO/IEC, 2001]) and reusability [Lincke and Löwe, 2007]. External attributes are important because knowing or predicting them plays an important role in quality assurance. Some internal attributes are correlated with external attributes [Lincke and Löwe, 2007], therefore their metrics can be used as *indirect measures* for external attributes. Figure 2.1.1 gives an overview the relations between quality properties (external attributes) and metrics measuring

internal attributes. The descriptions of the metrics are provided in Appendix A. As we can see in the figure, each quality property can be divided into a set of sub-properties. Their definitions can be found in Appendix B.

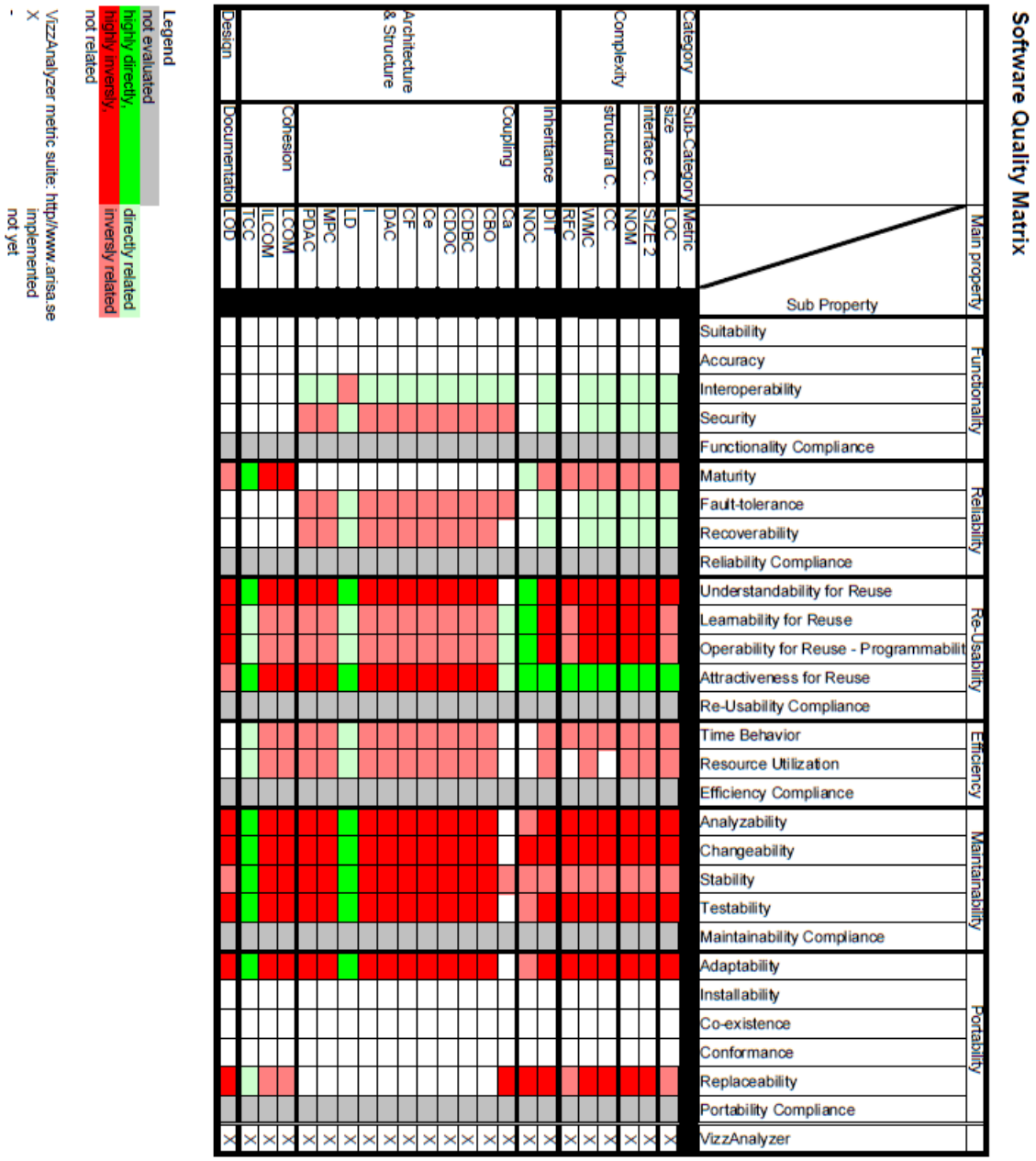

Figure 2.1.1. Relations between quality properties and metrics [Lincke and Löwe, 2007]

Available online at: http://www.arisa.se/compendium/quality-metrics-matrix.pdf

Linked from: http://www.arisa.se/compendium/node1.html

For some quality properties it is possible to find *surrogate measures*. For example, efficiency can be expressed by the execution time required for particular functions [Scotto et al., 2006], and maintainability by the time taken for a specified maintenance task [Georgiadou et al., 2003]. However, such measurement is often possible very late in the lifecycle [Georgiadou et al., 2003], when the software is ready for use.

Product metrics are dependent on the programming paradigm. In the procedural paradigm metrics measure functions (procedures) and their interactions, while in the object-oriented paradigm – classes and their interactions [Li, 1999]. Procedural metrics are for instance size in Lines of Code (LOC) and McCabe's Cyclomatic Complexity [McCabe, 1976] – they measure procedural structures. Examples of object-oriented metrics are Weighted Method Count (WMC), Depth of Inheritance Tree (DIT), and Coupling Between Objects (CBO) – they apply to entities specific to object-oriented programming. Object-oriented metrics cannot be applied to procedural programs, but procedural metrics can be applied to functions (methods) of the classes of an object-oriented program. LOC can also be extended to the whole system.

2.2. Agile software development

Agile software development is a group of software development methods (methodologies) based on human-oriented approach and light-but-sufficient practices [Cockburn, 2002]. The aim is rapid and effective production of valuable software with constant support for requirements changes while maintaining simplicity of the development process by elimination of any unnecessary activities. Because of the emphasis on simplicity agile methods are also called light or lightweight. They were created with the belief that they constitute better ways of developing software [Beck et al., 2001], as opposed to traditional development methods, such as the waterfall method. Agile methods

advocate starting with a small set of requirements and extending it in an iterative manner, as with time the customer's and the developers' understanding of the system improves. In contrast, traditional methods demand early availability and stability of all requirements and, in addition, many formal process artifacts, therefore from the agile point of view they diminish the customer's competitive advantage and impose superfluous burden on the developers.

The first acknowledged agile methodology was Extreme Programming (also know as XP) introduced in 1999 [Beck, 1999; Beck, 2000]. Since then many other light methodologies were invented and rediscovered [Abrahamsson et al., 2002]. In February 2001 a group of 17 software practitioners and consultants, most of whom were authors or co-authors of lightweight methodologies, met in an attempt to find a common denominator for the various existing light methods. They agreed that the word agile was the best fitting to describe the character of the methods [Cockburn, 2002], and wrote the Manifesto for Agile Software Development, called the Agile Manifesto for short. The Manifesto contains four core values and 12 principles of agile software development. The four values are [Beck et al., 2001]:

individuals and interactions over processes and tools,

working software over comprehensive documentation,

customer collaboration over contract negotiation,

responding to change over following a plan.

The values are presented in a relation to other values which are less important from the agile point of view (but not completely unimportant). The principles are statements on a more detailed level that are consistent with the four values.

Some of the best known agile methodologies are: the above-mentioned Extreme Programming, Scrum [Schwaber, 1995; Schwaber and Beedle, 2001],

Crystal family of methodologies [Cockburn, 2002], Feature Driven Development (FDD) [Palmer and Felsing, 2001], the Rational Unified Process (RUP) [Kruchten, 1996; Kruchten, 2000], Dynamic Systems Development Method (DSDM) [Stapleton, 1997], and Adaptive Software Development (ASD) [Highsmith, 2000]. This selection comes from the first systematic review of agile software development methods by Abrahamsson et al. [2002] (with Open Source Software development excluded, as it is not exactly a methodology). At the time it was published there was no agreement on the meaning of the concept of agility. The authors provide their own definition: agile software development is incremental, cooperative, straightforward, and adaptive. This definition, however, does not refer to the Agile Manifesto, nor does it reflect all its ideas. Therefore, in this thesis we will use another definition, formulated by Mnkandla and Dwolatzky [2007] and inspired by a conference article by Lindvall et al. [2002], which says that agile software development methods are:

1. Iterative – a software problem is solved by finding successive approximations to the solution starting from an initial minimal set of requirements. Design changes with each release as the requirements are updated.
2. Incremental – the system is partitioned into small subsystems by functionality and new functionality is added with each new release.
3. Self-organizing – the team is given the autonomy to organize their work.
4. Emergent – which means that
   (a) the system is allowed to emerged from a series of increments,
   (b) a method of working emerges, based on the self-organization,
   (c) a framework of development technologies will emerge as the other two emerge.

The value of agility is in allowing the above concepts to mutate within the parameters set by the agile values and principles [Mnkandla and Dwolatzky, 2007]. Such an exhaustive definition allows for generalization over agile methods, without having to conduct a detailed analysis for each of them separately.

Of course, there are some differences between agile methods that must be taken into account. One essential difference is the extent to which the development lifecycle is covered. Another is the supported project aspects: management, process, or practices. These differences are depicted in Figure 2.2.1. In the figure, each method is divided into three bars. The top bar indicates whether a method provides support from project management. The middle bar indicates whether a process through which the software production proceeds is described. The bottom bar shows whether the method offers concrete guidance in a form of specific practices, activities, or work products. The length of the bars shows which phases of the development lifecycle are supported. [Abrahamsson et al., 2003]

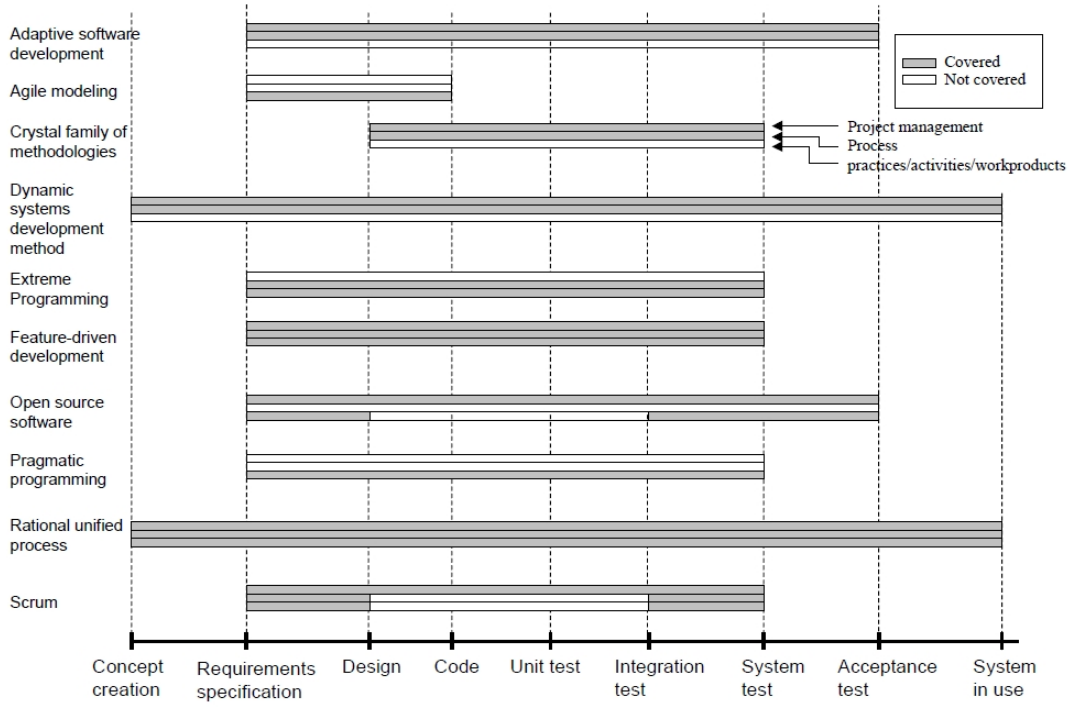

Figure 2.2.1. Software development lifecycle support [Abrahamsson et al., 2002, p. 95]

As it can be seen from Figure 2.1.1, most of the agile methods are neither extensive nor precise. RUP is an exception, but it is a commercially sold development environment, so it is not available to everyone. DSDM covers two of the three discussed aspects throughout the whole lifecycle, but it is available only to the members of the DSDM consortium. The other methods need complementary approaches in some regards [Abrahamsson et al., 2002].

Some methods do not offer any concrete guidance (practices/activities/work products) and rely only on abstract principles [Abrahamsson et al., 2003]. Those that do, often recommended different practices, so there is no unified set of agile practices. However, some practices are more universal and popular than others. Some best-known ones are: pair programming, test-first development, refactoring, and customer involvement. They have been shown by some studies to positively affect product quality [Dybå and Dingsøyr, 2008; Sfetsos and Stamelos, 2010]. This is the essence of agile quality assurance. However, it has not yet been scientifically proved that agile practices assure product quality [Concas et al., 2008], or that agile quality assurance is superior to that of conventional methods [Berki et al., 2007]. For that, more studies with rigorous and systematic approach, including formal measurement, are needed.

3. Literature review

There are rather few references on software metrics used in agile software development, especially product metrics. The ones that do exist are not overly comprehensive – most of them are concerned with general guidelines for agile measurement without suggesting specific metrics, or concentrate on a single aspect of agile software development related to metrics, or present a case study.

3.1. General concerns

Metrics are not a part of the agile paradigm and incorporating them into an agile process is not a trivial task. Since the idea of agile software development is freeing the process from burdensome practices and concentrating directly on producing working software, which is the primary measure of progress [Beck et al., 2001], it may even seem counterproductive to introduce a practice that requires additional documentation-related work. Indeed, in agile methods an extensive metrics program is unlikely to be used, as in traditional metric programs progress is measured by conformance to plans [Siakas et al., 2005]. On the other hand, some process monitoring is necessary in order to know it is working well. This is probably why Hartmann and Dymond [2006] state that ongoing measurement, whether anecdotal or formalized, is inherent in the agile approach. Suitable metrics provide holistic insights into the strengths and weaknesses of the process and product, assuring general stakeholder satisfaction [Berki et al., 2007]. Therefore, metrics are not only a tool for process monitoring, but also for process improvement, as well as for product improvement [Siakas et al., 2005]. The effort needed for the collection and recording of metrics can be minimized with automated tools [Berki et al., 2007; Layman et al., 2004; Scotto et al., 2006].

The existing quality standards applied in traditional software development, for example ISO 9126, are said to be incompatible with agile methods [Kunz et al., 2008], with the metrics often being vague, irrelevant, or unsuitable [Hwong et al., 2007]. Unfortunately, the authors of these claims do not explain further why that is the case, how exactly traditional quality metrics have failed to serve their purpose. Possibly the problem lies not in the quality standard itself, but in the ISO 1926 metrics, as they are reported as imprecise and therefore unusable also for a non-agile project [Bøegh, 2006]. Otherwise, it is somewhat hard to deduce, after all the software itself is not agile, only the process. Perhaps basing on that observation, a supposition antithetic to those of Kunz et al. [2008] and Hwong et al. [2010] comes from Berki et al. [2007], saying that agility probably lies in the speed and method of collection and not in the metrics themselves. Moreover, a systematic literature review by Sfetsos and Stamelos [2010] evaluates the findings of 46 empirical studies regarding quality in agile practices according to the ISO 9126 standard; the authors do not mention anything about the standard being inappropriate for agile methods.

It is, however, possible that some product metrics will behave differently in agile development than in traditional development. The study by Olague et al. [2007] can serve as an example. It is an empirical validation of metric suites to predict fault-proneness in object-oriented components in agile (or highly iterative) software development. It reveals that although the Chidamber-Kemerer (CK) metric suite [Chidamber and Kemerer, 1994] is a good and reliable predictor of fault-proneness, it will not remain effective forever. The metric suite will predict fault-proneness of classes effectively only as long as the classes are being changed and new classes are introduced, before the code base stabilizes and the dynamic nature of the development process subsides, which usually happens after several iterations.

Inappropriate metrics are also mentioned by Hartmann and Dymond [2006], yet again without specific examples or explanations. They do say, however, what good agile metrics should be like. A prerequisite is to understand the difference between two activities: measuring to deliver value and gathering data about the effects of the agile approach. Then there are ten heuristics to follow:

1. Metrics should be designed by people who understand the agile paradigm, so that they reinforce agile principles, and not the contrary (for example overtime, utilization percentage, paperwork).
2. Measure outcome, not output – outcome is measured with the delivered customer value, which can be maximized while the planned output is reduced.
3. Follow trends, not numbers – data should be aggregated to the appropriate level.
4. The set of metrics should be small, in keeping with the "just enough" agile approach, as too much information may actually obscure important trends.
5. Metrics should be easy to collect – automated tools, that are themselves easy to use, are needed.
6. Metrics should reveal their context and significant variables.
7. Metrics should inspire meaningful conversations, as face-to-face conversations are a tool for process improvement.
8. Feedback from metrics should be provided on regular basis, and be available especially for management meetings and iteration retrospectives. This corroborates Leffingwell's [2006] opinion that feedback from metrics should primarily be analyzed during iteration and

release retrospectives, in order to support reflection and adaptation of the team.

9. A metric may measure the product (value) or process. Also the appropriate audience for each metric should be considered, and the metric's context and assumptions documented, in order to encourage its proper use.
10. A good metric encourages "good-enough" quality, which is defined by the customer or other business representative, not the developers.

The authors emphasize the importance of the customer value in the software product. They support that *Business Value Delivered* should be the one key metric driving the whole organization, and all other metrics should be regarded only as tools helping to improve the key metric. Business value cannot be measured only in terms of working software, economic factors have to be taken into account as well, therefore collaboration across organizational units is required.

Ktata and Lévesque [2010] draw on Hartmann and Dymond's work in order to develop an approach to designing a measurement program for Scrum teams. The guidelines they provide are rather general and are predominated by the use of Goal-Question-Metric (GQM) method [Basili and Weiss, 1984] for gradual introduction of metrics which appear to be relevant to those involved in the program. The issues that were found to need the most attention are business value visibility, technical debt, and estimation.

3.2. Size and quality

According to IEEE guidelines on basic metrics, you should measure product size and quality, at minimum [Van Schooenderwoert, 2006]. There are numerous ways to measure size, for example lines of code (LOC), function points, or story points, all of which are quite typical. Different metrics will have different effects on the project. Dubinsky et al. [2005] give an example of an

unusual way to measure size: by test points. One test point is one step in an automatic acceptance test scenario [Talby et al., 2005] or one line of unit tests. The amount of acceptance tests for a given feature is usually proportional to the feature's size and complexity [Dubinsky et al., 2005]. The metric was used to present the amount of completed work. This is more strict than in the agile paradigm, where the measure of progress is working software. Here it is not enough that it is working, it also has to be covered by automated tests. That motivated the developers to write more tests, and in turn the thorough tests allowed an early discovery of the inserted defects. Van Schooenderwoert [2006] had a similar observation: agile teams find defects very fast during unit testing, so much that it is not practical to record them, which results in fewer defects inserted to the code base upon the integration.

The test points metric could be seen as an extension of the Running Tested Features (RTF) metric [Jeffries, 2004]. RTF is a number of end-user features (small, cohesive pieces of functionality visible to the end user) that are working, continuously passing all automated acceptance tests provided by the requirement givers (the customers), and have been released. It is used for tracking progress. Jeffries asserts that if the development team is required to make the RTF metric grow steadily and smoothly throughout the project duration from the beginning (first or second week), they will be driven to work in an agile manner. This is because of the metric's requirements:

- Growth from the beginning requires the focus on features, not design documents.
- Continuous growth requires early and often integrations.
- Testing with independent acceptance tests requires constant contact with the customer.
- Continuous testing requires test automation.

- Smooth growth requires keeping the design sound with refactoring.

If instead of the number of the features we count the sum of the test points of each feature, we get the test points metric. This way we can additionally track the size while keeping the mechanism of action of RTF and thus all of its benefits.

Quality is often regarded as absence of defects, and as such can be measured by the cumulative number of defects and defect density (number per KLOC (=1000 LOC)). The cumulative number of defects is an indication of testing effectiveness (if testing is not effective, we cannot trust that the value of defect density is correct), and defect density indicates product quality (the lower the defect density, the higher the product quality). It is especially important to control the number of defects delivered to the customer, as customer satisfaction is the highest priority in agile software development [Beck et al., 2001]. Defect density that exceeds a certain level in a customer-delivered product is considered malpractice [Van Schooenderwoert, 2006]. It is also beneficial to record the defect type and root cause and calculate the distribution of defects per origin and per type. This provides a quantitative assessment to the management on where to invest the quality assurance resources [Bartels et al., 2009], which is very important, as poor effort allocation is one of the major root causes of rework and poor project performance [Yang et al., 2008]. Root cause analysis can help decide how to best prevent more occurrences of a defect [Van Schooenderwoert, 2006].

Quality can also be measured indirectly, as there are metrics correlated with quality properties (see section 2.1), for example the CK metric suite. Such quality measurement in agile software development could benefit from the use of *kanban guards* [Heidenberg and Porres, 2010]. The concept involves viewing the development as a set of queues of activities. The queues require appropriate

management, so that no queue becomes congested. A kanban guard is an extension of a metric with additional components: an interpretation function and an advice function. The interpretation function maps the input to a value between 0 and 1, and also divides the output value range into three sectors marked by colors: green, orange, and red. An output value in the green sector means that no action is required, in the orange sector that there is a problem in the system which requires some action, and in the red sector that there is a severe problem that requires immediate attention and probably stopping the current activities in order to fix the problem. The advice function provides information on how to handle the particular situation, for example to create a new task and add it to a task queue, to give priority to certain types of tasks, or to stop the execution in one queue until congestion in its upstream successors has been removed. Kanban guards monitor the software artifacts and indicate when human attention is needed.

3.3. Agile practices

Kunz et al. [2008] propose to combine software measurement with refactoring in order to indicate when a refactoring step is necessary, how important it is, how it affects quality, and what side effects it has. Metrics would be used here as triggers for needed refactoring steps. The suggested approach for defining the metrics is to use the GQM and/or the CK metric suite with possible modifications. The values of the chosen metrics should be normalized and have assigned thresholds for interpretation on the basis of empirical data or a chosen quality model. There is a tool that implements this approach, UnitMetrics [UnitMetrics, 2007]. It features a metric set based on CK metrics and the work of Henderson-Sellers [1996], Bloch [2001], and Martin [2003], among others. Apart from automated measurement, UnitMetrics also provides interpretation and presentation functions (diagrams and visualizations). Such a tool could be

supplemented with kanban guards [Heidenberg and Porres, 2010] (see section 3.2). The output value of the interpretation function in the green sector would mean that the design quality is high and no refactoring tasks are needed, in the orange sector – that the quality is too low and a refactoring task should be created and added to the refactoring queue, and in the red sector – that the current development task should be stopped and refactoring tasks should be performed instead. This way, the need for refactoring can also be justified to the project manager and/or the customer, who may not recognize its importance, as it does not change the functionality of the system. Metrics are an objective evidence of code quality, and it is rather understandable that lower code quality will cause lower maintainability. On the other hand, too much refactoring, which is a waste of time and not in line with the agile light-but-sufficient (or "just enough") philosophy, is also prevented by kanban guards.

Metrics not only can aid the execution of agile practices, but also provide data that reveals how agile practices affect software quality. A study of a medium-sized agile project featuring a web application, reported by Concas et al. [2008], has shown an improvement in quality metrics (the CK suite) when agile practices (pair programming, test-driven development, and refactoring) were applied, and a significant decline when the practices were discontinued. Although no definitive conclusions can be drawn from the observation of a single project, this study is the first step towards more rigorous and systematic assessment [Concas et al., 2008].

3.4. Requirements and design quantification

Tom Gilb strongly believes that quantification of requirements is an essential concept missing from the agile paradigm, or even from software engineering in general. He claims that this lack is a risk for project failure, as software engineers and project managers cannot properly manage project results, control

risks and costs, or prioritize tasks [Gilb and Cockburn, 2008]. Especially important are quality characteristics, because "functions and use cases are far less interesting" [Gilb and Cockburn, 2008] and "that is where most people have problems with quantification" [Gilb and Brodie, 2007]. Gilb assures that only numerically expressed quality goals are clear enough and therefore quantification is a step needed on the way from high level requirements to design ideas.

Design ideas also should be quantified [Gilb and Brodie, 2007]. It is done by the estimation of their value (to the customer – business value, not technical) and cost (effort). Then it is possible to identify the best designs – the ones with the highest value-to-cost ratio – and reprioritize the following development steps. A similar idea is described by Kile and Inampudi [2007]. The value and cost for implementing a requirement is estimated (without considering different design ideas) and the requirements are prioritized according to the result. This was a solution to a problem with the requirements prioritization ability of the customer and the development team. With some requirements having high value and high cost, and others having moderate value but very low cost, it was not easy to compare their desirability without quantification.

Berki et al. [2007] criticize the idea by noting that agile metrication procedures could be difficult to establish for the requirements of agile software development. Indeed, Gilb [2007] himself admits that a development team described in his case study found it hard to design useful metrics. It is his belief that all qualities can be expressed quantitatively and that it is important to believe that everything can be measured, but no practical solution is offered. More critique comes from Alistair Cockburn [Gilb and Cockburn, 2008], who objects the idea of numbers being necessary for the clarity of requirements. According to Cockburn, communication and feedback are far more important

and allow true understanding of the requirements and the reasons for them. Another reflection is that there is no evidence for quantified quality requirements having a demonstrable return on investment (ROI) for increasing stakeholder value.

4. Findings

The finding are divided into two categories, each described in a separate section. The first section presents the findings on measurement in general by answering why, how, and when to measure, and who should measure and for whom. In the second section the same questions are applied to specific metrics suggested to use in agile software development, which results in more detailed guidelines for each metric or metric type.

4.1. General

4.1.1. Why to measure?

There is general agreement on the importance of metrics. However, the range of reasons provided is quite wide. These reasons can be divided into two categories: product quality management and development process management.

Using product metrics to manage the product quality is rather self-evident, since it is information on the product quality that they provide. It is said that measurement allows improvement [Hartmann and Dymond, 2006], and even that it is necessary for improvement ("If you cannot measure it, you cannot improve it" – William Thomson, later Lord Kelvin (1824-1907); DeMarco [1982], cited in Berki et al. [2007]). What is certain is that measurement is an objective way to assess improvement. For instance, refactoring, which is an essential agile practice, is meant to improve the internal structure of the source code without changing its functionality and therefore to enhance maintainability; however, product metrics often reveal that it has opposite results [Bryton and Brito e Abreu, 2009]. Comparisons before and after refactoring could be a particularly fruitful results of using product metrics [Berki et at., 2007]. Similarly, the effect of other agile practices on the quality of

the product can be observed with the help of metrics. Beneficial and harmful behaviors can be identified and reinforced or eliminated. This way, product metrics also contribute to process improvement.

Metrics can also aid the application of agile practices. For example, in refactoring they can give information on the appropriate time for and the significance of a refactoring step [Kunz et al., 2008].

Metrics detect defects and potential problems, and Kunz et al. [2008] affirm that early observation of quality is crucial to maintain the software's stability throughout an evolutional development process (agile processes being a type of evolutional processes). The later a defect is detected, the more effort is needed for tracing and fixing it [Boehm, 1981]. Therefore, early defect detection and prevention of errors later in the lifecycle decreases overall development costs, which on the organizational level means enhanced efficiency and productivity. This enhances the reputation of both the organization and its business practices [Ordonez and Haddad, 2008]. Moreover, incorrect estimates of code quality may cause significant financial losses [Harjumaa et al., 2008].

The highest priority in agile software development is to satisfy the customer through the delivery of valuable software [Beck et al., 2001]. This fundamental principle means agile transition entails a shift in the focus from cost to value [Hartmann and Dymond, 2006]. The value is in working [Beck et al., 2001], competitive [Gilb and Brodie, 2007], usable, marketable and selling [Hartmann and Dymond, 2006] software. Without measurement it is not possible to directly control the delivery of these benefits to the customer and other stakeholders (for example, users). Quantification of requirements, design, and feedback [Gilb and Brodie, 2007] produces data necessary for the evaluation of various design ideas and choosing the one that fulfills the requirements to the highest degree, and therefore provides the most value.

Measurement, as a tool for supporting the management of the development process (not only agile), provides insight into the strengths and weaknesses of the process and product [Berki et al., 2007]. This allows evaluation of the project status (stability, fitness for beta testing and delivery) and early risk identification, and thus motivates change in order to ensure successful completion [Atkinson et al., 1998]. Following on an idea from Stark et al. [1994], Fenton and Neil [2000] specify that the support metrics provide should be mainly focused on quantitative managerial decision-making during the software lifecycle, because that is their most important objective. It has been found that increased use of metrics in decision-making leads to increased organizational performance, and vice versa [Gopal et al., 2002]. Bartels et al. [2009] go one step further and assert that measurement is a *primary* tool for managing software lifecycle tasks, such as planning, controlling and monitoring of project plans, especially in agile organizations, where these activities are performed daily.

4.1.2. What to measure?

There are hundreds of existing product metrics. How to choose the ones suitable for a particular agile development project, considering that not everything that can be measured, should be [Hartmann and Dymond, 2006]? Which metrics are aligned with the agile values and principles? Or does agility only affect the way measurement is performed, as suggested by Berki et al. [2007]: "(...) agile metrics are not themselves very different from traditional metrics (...). The difference in 'agility' probably refers to the method and speed of collection."?

It is impossible to create a metric set that would suit all agile projects. Every project has different goals and needs, and, as noted in section 2.2, the incremental and emergent nature of agile methods [Mnkandla and

Dwolatzky, 2007] implies that metrics – as part of the framework of development technologies – should also be allowed to emerge. There are, however, methods for choosing suitable metrics.

In order to decide what to measure one can use a traditional, top-down approach called the Goal-Question-Metric (GQM) method [Basili and Weiss, 1984]. It starts with stating a business goal (or a set of goals), which is then refined into a set of questions, and each question is used to inspire the formulation of a metric. The measured values undergo interpretation in order to answer the questions and help achieve the goal. This approach is also called goal-driven measurement. The GQM method is used by Bartels et al. [2009], for example, for the goal of controlling the number of defects delivered to the customer. The formulated questions and metrics are shown in Figure 4.1.2.1.

1. Where is the origin of defects?	1. distribution of defects per origin
2. How effective is the testing?	2. number of defects / number of bugs
3. How many defects are injected?	3. number of defects injected per sprint
4. What types of defects are injected?	4. what types of defects are injected

Figure 4.1.2.1. Exemplary questions (left) and metrics (right) in the GQM method

The authors claim that these metrics are very effective for an agile organization. GQM is also advocated by Ktata and Lévesque [2010] as suitable for agile development. The aspects of a software product that should be measured were found to be the visibility of business value, the visibility of technical debt and test coverage (the number of tests in each level of testing and area of the software).

Other mechanisms for finding measurable goals, though not as noted as GQM, are for example the Quality Function Deployment approach [Kogure and Akao, 1983] and the Software Quality Metrics approach [Boehm et al., 1976; McCall et al., 1977]. Because effective measurement must be focused on specific goals [Basili et al., 1994], an opposite, bottom-up approach is considered

unsuitable for determining the metric set. Furthermore, a bottom-up approach entails measurement of a large number of metrics, which most like would decrease the quality of data and become burdensome for the personnel [Harjumaa et al., 2008].

Some product metrics have been directly mapped to software quality properties from ISO/IEC 9126 standard, with indication of the correlation strength [Lincke and Löwe, 2007]. This can be helpful when choosing metrics to answer questions regarding software quality in the GQM method. Another way for choosing quality metrics is to use a preexisting metrics suite (or several of them), where the main goal and the benefits are outlined, and the meaning of each metric and the interpretation of results are provided by the authors. For instance, the Chidamber and Kemerer (CK) suite of quality metrics [Chidamber and Kemerer, 1994] is the most validated in the literature for object-oriented systems [Concas et al., 2008; Ambu et al., 2006]. Since agile methodologies use mainly object-oriented technologies [Siakas et al., 2005], CK metrics are largely applicable to agile software development. The CK suite consists of six metrics (based on [Concas et al., 2008]):

- **Weighted Methods of a Class (WMC):** A weighted sum of all the methods defined in a class. Chidamber and Kemerer suggest assigning weights to the methods based on the degree of difficulty involved in implementing them [Chidamber and Kemerer, 1994]. In the simplest case, when all weights are unity, it is the same as the number of methods. WMC is a measure of the complexity of a class.
- **Coupling Between Objects (CBO):** A count of the number of other classes with which a given class is coupled. To be more precise, class A is coupled with class B when at least one method of A invokes a method

of B, or accesses a field (instance or class variable) of B. CBO denotes the dependency of one class on other classes in the system.

- **Response For a Class (RFC):** A count of the methods that are potentially invoked in response to a message received by an object of a particular class. It is computed as the sum of the number of methods of a class and the number of external methods called by them. RFC is both a measure of the complexity of a class, and of the potential communication between the class and other classes. In fact, in empirical measurements, RFC is often found to be correlated with both WMC and CBO – though WMC and CBO are not correlated with each other.
- **Lack of Cohesion in Methods (LCOM):** A count of the number of method pairs with zero similarity, minus the count of method pairs with non-zero similarity. Two methods are similar if they use at least one shared field (for example they use the same instance variable). LCOM measures the cohesiveness of methods within a class.
- **Depth of Inheritance Tree (DIT):** The length of the longest path from a given class to the root class in the inheritance hierarchy. DIT is the total number of superclasses of a given class, at all levels, and measures how many classes can influence the class through the inheritance mechanism.
- **Number of Children (NOC):** A count of the number of immediate subclasses inherited by a given class. It is a measure of the width of the inheritance tree whose root is the class.

CK metrics predict fault-proneness of classes [Basili et al., 1996] and maintenance effort [Li and Henry, 1993]. Higher values of CK coupling and cohesion metrics are associated with reduced productivity and increased rework/design effort [Chidamber et al., 1998]. In general, the lower the value of

CK metrics, the better the software quality [Concas et al., 2008]. The CK metric suite can be used to aid refactoring, as described in section 3.3.

In the procedural paradigm fault-proneness can be evaluated by combining size (for example LOC) and complexity (for example McCabe's Cyclomatic Complexity [McCabe, 1976]). Large modules with high complexity tend to have the lowest reliability [Ordonez and Haddad, 2008].

An essential part of product quality management is the control of defects. As mentioned in section 3.2, useful metrics in that regard are: cumulative number of defects, defect density, origin and type of each defect, and the distribution of defects per origin and per type. Defect density requires the measurement of the system size (for example in LOC or function points). Size is often used for calculation of different metrics. A quite new, agile approach to size measurement is to use test point, which has the benefit of reinforcing behaviors desired in agile software development (see section 3.2).

Product quality metrics can also provide information that can be useful for project management. There is a metric suite designed particularly for that purpose. Atkinson [1998] proposes a set of four metrics that together represent the project's status and indicate whether it is moving toward success or failure. This set includes cumulative effort (hours/week), source code growth (KLOC), test suite development (number of generated test suites), and defect found/fixed rates. The cumulative effort is an indicator of the work being accomplished, and it should be at a reasonable level from the beginning – early lack of effort is a sign of difficulty and potential failure. The source code growth should be rapid in the initial implementation phase and stabilize in the final phase. A code base still growing at the end of the project is a significant risk. The test suite development should grow continuously and the number of tests should at all times be higher than the number of defects. The number of defects should grow

as the source code base grows or changes, otherwise it may indicate that the level of testing activity is insufficient. However, it should stabilize toward the end, as this will mean that most defects have been found and fixed. Atkinson claims that if the metrics follow their desired trends, called *signatures of confidence*, it means the project is highly likely to succeed, whereas otherwise it is at risk of failure. Figure 4.1.2.2 depicts the signatures of confidence for code growth, test suites, and defects found and fixed.

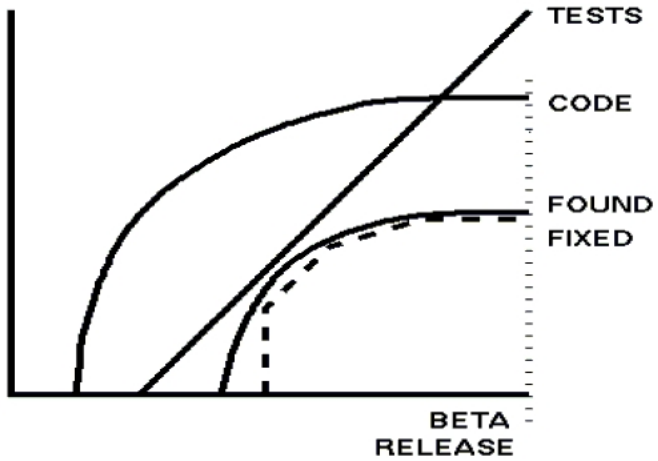

Figure 4.1.2.2. "Signatures of confidence" [Atkinson et al., 1998]

Another aspect of management that can benefit from metrics are decisions on the order of feature implementation and the acceptance or rejection of the proposed design ideas. Estimating value and cost and then calculating the value-to-cost ratio for requirements helps to prioritize them (the higher the ratio, the higher the priority) and for designs helps to choose the best design idea – the one with the highest ratio (see section 3.4).

4.1.3. How to measure?

Agility, defined as in section 2.2, has an immense impact on how measurement should be performed and utilized. It demands that all values and principles (see Figure 4.1.3.1) from the Agile Manifesto [Beck et al., 2001] are followed, which means the approach must be very different from traditional methods, because

agile methods “have certainly changed the way software development is done” [Mnkandla and Dwolatzky, 2007].

1. Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
2. Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.
3. Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.
4. Business people and developers must work together daily throughout the project.
5. Build projects around motivated individuals.
6. Give them the environment and support they need, and trust them to get the job done.
7. The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.
8. Working software is the primary measure of progress.
9. Agile processes promote sustainable development.
10. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
11. Continuous attention to technical excellence and good design enhances agility.
12. Simplicity--the art of maximizing the amount of work not done--is essential.
13. The best architectures, requirements, and designs emerge from self-organizing teams.
14. At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

Figure 4.1.3.1. Agile principles [Beck et al., 2001]

Hartmann and Dymond [2006] provide a set of rules for agile measurement (see Figure 4.1.3.2) that are in keeping with the agile principles. Heuristic 1 confirms that metrics have to be aligned with agility. It also mentions lean development, which by some is considered to be a part of the agile paradigm, or at least closely related to it. Several heuristics correspond to principle 10: heuristic 2, because outcome is the value for the customer and it can be achieved by actually reducing the output (maximizing the amount of work not done);

heuristic 4 – too much data can obscure important trends; heuristics 5 and 10 (for quite obvious reasons). Heuristic 7 is in line with principle 6 and 12, and with the first value ("Individuals and interactions over processes and tools") because it shows that metrics are just a tool and their real meaning is in their usage by people. Heuristic 8 reinforces the fundamental agile concepts – iterative and incremental process.

A good metric or diagnostic:

1. Affirms and reinforces Lean and Agile principles.
2. Measures outcome, not output.
3. Follows trends, not numbers.
4. Belongs to a small set of metrics and diagnostics.
5. Is easy to collect.
6. Reveals, rather than conceals, its context and significant variables.
7. Provides fuel for meaningful conversation.
8. Provides feedback on a frequent and regular basis.
9. May measure Value (Product) or Process.
10. Encourages "good-enough" quality.

Figure 4.1.3.2. Heuristics for wise agile measurement [Hartmann and Dymond, 2006]

Since agile methods advocate minimal documentation ("Working software over comprehensive documentation"), it is important that measurement does not cause overhead in this aspect. The solution would be automated data capture [Berki et al., 2007; Layman et al., 2004; Scotto et al., 2006], which implies the necessity of using automated tools, including reporting features, so that the reporting process is also as lightweight as possible. This idea is also advocated by Scott Ambler [2005]. He believes that metrics collection should be automated as much as possible, and if any manual measurement is necessary, it should be simple and "good enough", meaning that it does not have to be perfect or overly accurate. Many other authors also are convinced that automated tools are essential, whether in conventional or

agile development, not only because they reduce the amount of extra work, but also because they help ensure the validity and integrity of the collected data [Daskalantonakis, 1992; Hall and Fenton, 1997; Harjumaa et al., 2008; Iversen and Mathiassen, 2000; Offen and Jeffery, 1997; Paulish and Carleton, 1994; Pfleeger, 1993].

Ambler [2005] additionally offers two other rules, which together with automated measurement constitute the basics of agile metrics collection and reporting: communication and feedback. Communication is necessary to understand the meaning of and the reasons for the measurement results, because a metric in and of itself cannot provide such information, only the people involved can. Feedback must be rapid, so that appropriate actions follow as soon as possible. It is important that feedback is acted upon, otherwise the value of measurement is questionable [Niessink and van Vliet, 1999]. That especially concerns negative feedback and corrective actions, and it is advisable to have an action plan prepared beforehand for such cases, otherwise it is likely that nothing will be done [Niessink and van Vliet, 1999].

A general rule that also applies to agile measurement is that it is important to be aware of the purpose of a metric, otherwise metrics may not yield the expected benefits. Confusing an effort-predicting metric with a defect-predicting metric, for instance, can nullify the metric's usefulness [Beizer, 1990]. Moreover, if the development team members do not understand how metrics will benefit the project, they may not believe in their usefulness and perceive them as overhead, which in itself is undesired in agile software development, and additionally it can cause adverse effects such as resistance to metrics activities and compromised data integrity [Umarji and Seaman, 2009]. Inversely, the better the developers recognize the purpose and importance of the data, the better is the quality of the data [Iversen and Mathiassen, 2000]. That importance

should also be visible in practice – individual developers should be aware that the data they provide is appreciated and used [Harjumaa et al., 2008].

In addition to understanding the purpose of individual metrics, it is also essential to remember that a single metric contains only pieces of information that are parts of a bigger picture. The true value of metrics lies in evaluating them in terms of their context and the relationship with other data [Bundschuh and Dekkers, 2008].

Trust is of great importance in agile software development (Agile Manifesto, principle 6). Therefore, confidentiality of measures should be maintained and not all results should be available to everyone. Particularly access to measures of individuals should be restricted only to those to whom the data is directly related, and wider availability must be agreed upon. Data should also never be used directly against developers or other reporters. [Harjumaa et al., 2008]

4.1.4. When to measure?

Of all the metrics-related questions in agile software development, the one about timing seems to be the least considered. Authors often do not mention anything on the subject, and if they do, they are rather general and brief about it. For example, Ordonez and Haddad [2008] merely state that metrics can be used at each state of software development, without going into details. Similarly, Gilb and Brodie [2007] say: "Quantification must be utilized throughout the duration of an agile project", and only list the activities where quantification is used (stating requirements, driving design, assessing feedback, and tracking progress). Although these activities can be mapped on to the stages of a project, more specific guidelines are needed. Hartmann and Dymond [2006] mention in their heuristics for wise measurement that metrics

should provide feedback on a frequent and regular basis and that the data should be available at each iteration retrospective.

Gilb and Brodie's statement can be supported by the illustration of quality in the lifecycle by Suryn et al. [2002] (Figure 4.1.4.1). The objective of software development is for the product to have the required effect in a particular context of use. As the software process influences the product quality, in order to produce these effects, measurement and evaluation of the quality of software product has to be present during all its lifecycle [Suryn et al., 2002]. In order to provide timely feedback for corrective actions, data should be collected, validated, and analyzed in real time [Ktata and Lévesque, 2010].

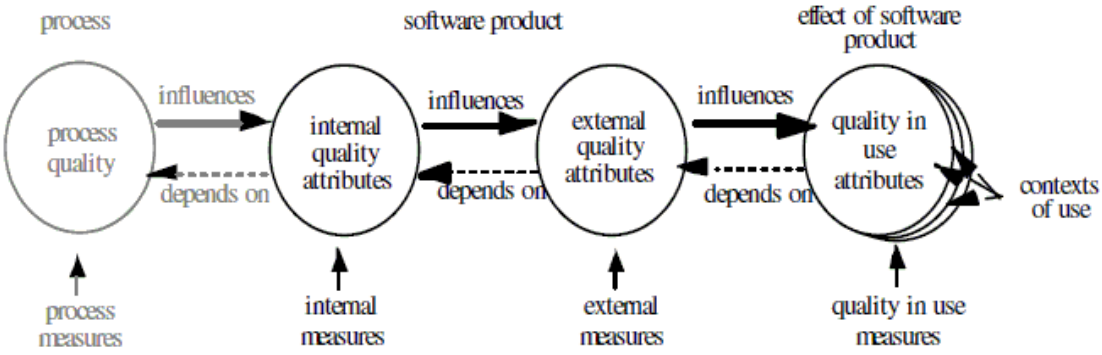

Figure 4.1.4.1. ISO/IEC 9126 Quality in lifecycle [Suryn et al., 2002]

However, what does "real time" mean exactly? If data is recorded automatically, developers may feel that their privacy is threatened [Johnson et al., 2005], which could compromise the environment of trust important in agile development. Therefore, data for a wider audience should be extracted from the code repository, rather than from individual developers' workstations, but it should be done regularly – at every code check-in or during the daily build and test procedure [Heidenberg and Porres, 2010].

There is one article [Leffingwell, 2006] solely dedicated to the topic of the timing of measurement in agile projects. However, it is not a scientific article. In it, Leffingwell argues that measurement should happen during iteration

retrospectives and release retrospectives. He proposes two categories of metrics: quantitative and qualitative. Quantitative metrics for an iteration consist of process metrics, such as the number of user stories accepted/rejected/rescheduled/added, and product metrics related to quality (defect count) and testing (number of test cases, percentage of automated test cases, unit test coverage). Quantitative metrics for a release measure the release progress with value delivery (number of features delivered to the customer and their total value expressed in feature points, feature debt – existing customer commitments), conformance to release date, and technical debt (number of refactoring targets and number of refactorings completed, also called architectural debt or design debt). The qualitative assessment for both iteration and release requires listing what went well and what did not, revealing what should be continued and what needs to be improved.

The question when to measure applies also to each metric separately. Different metrics may be needed at different stages of the lifecycle, some metrics may lose their effectiveness with time, like the Chidamber and Kemerer suite (see section 3.1), or even become misleading. Measurement should be continued only as long as it has some positive influence, for instance in development costs, developer productivity, or customer feedback [Harjumaa et al., 2008]. Therefore, it should always be planned right from the start when to stop using a metric [Hartmann and Dymond, 2006].

4.1.5. Who measures and for whom?

The measurement activities are performed by and for some stakeholders of the software project. A stakeholder is someone who has interest in the success of the project. In this thesis we consider only the *relevant* stakeholders, that is those who may directly benefit from the use of product metrics. The relevant stakeholders are:

- **Project manager.** Coordinates the project. Makes decisions on strategic, tactical and operational levels and controls that these decisions are followed [Siakas et al., 1997].
- **Developer.** Works directly with the software product and commits to its development. Responsible for requirements analysis, design, coding, and maintenance.
- **Tester.** Responsible for testing of the software product inside the organization where it is created. Testers and developers together form the development team.
- **Customer.** Person(s) or organization(s) that order the project and sponsor it, and their representatives.
- **User.** End-user of the software product, for whom the product is meant.

The question "who measures?" is not a very substantial issue, considering the involvement of automated tools. Usually there will be no need for anyone to perform manual measurement. Some activities may be semi-automated. For example, defects are often entered into a bug-tracking system, which may not include reporting features, and tools for data extraction from such a system may not exist, so manual reporting will be necessary. However, it is not measurement as such, but simply a matter of copying data from one place to another. This and – in the rare cases it is needed – manual measurement should be done by a person who needs the data the most. For example, if the distribution of defects per component is used by the project manager when making the decisions about quality assurance resource allocation, then it is the project manager that should be responsible for obtaining such data. It must be assured that everyone has access to the data they need. The customer may be an exception, as it may not be desirable for them to have direct access to any project data. In that case they are presented with reports especially prepared for

them. Usually it will be the project manager that creates the report, but if it appears suitable, it may be agreed that developers and testers also contribute to it. However, it is important to keep in mind the agile intent of not burdening the development team with work that is not a direct contribution to the development of the product.

Depending on their role, different people have different interests in the software, and therefore they focus on different metrics [Bundschuh and Dekkers, 2008]. The development team's interest is technical, the customer's interest is economic, and the project manager, being an intermediary between them, has to reconcile both (unless the role is divided between two people, one dealing with technical issues and the other one with economic ones – then they have to reconcile the two interests together). The ultimate goal is to create a product that will solve some problem the user wants to solve, and that will be chosen over any competing products – which probably will happen if the user considers the product to have the best value for money.

For whom the metric is meant depends on its purpose. Metrics for supporting managerial decision-making will be used mainly by the project manager, and also the development team, as agile teams are self-organizing and entitled to making decisions about their work. Managerial metrics may be shared with the customer to present the project's situation and rationale for decisions. Metrics indicating quality, on the other hand, are of interest to all stakeholders, but not equally, because different stakeholders have different perceptions of quality attributes [Siakas et al., 1997]. For example, maintainability is of a great importance to the developer, as it is correlated to the degree of rework. The customer is also concerned, because more rework means the project will take longer and possibly cost more (in case of an hour-based budget). The user has little interest in maintainability. The qualities that

matter to the user are those that can be experienced while using the software product – functionality, usability, efficiency, portability. In turn, these qualities are only indirectly relevant for the developer – only in the sense that they have to ensure their appropriate levels in the product. The customer's primary interest lies in the business value of the software product [Ktata and Lévesque, 2010]. In consequence, apart from the cost of the product development they are largely interested in the same qualities as the user, because providing a product suited to the user's needs increases the probability that the product will sell. Similarly, as the project manager's job is to ensure the customer's satisfaction, they value the same quality properties. The developer is primarily concerned with the amount of effort that working on the product requires [Siakas et al., 1997]. They want the software to be easy to change and extend, simple and understandable. Metrics showing which parts of the system comply with these requirements and which do not will aid the daily development work by identifying refactoring targets. Such information is also important to the testers, as the worse the code quality, the more defect-prone it tends to be. Therefore, quality metrics indicate where more testing effort is needed. The opposite is also true – high quality modules require less testing, which allows time-savings and reinforces the "just enough" philosophy of agile software development.

4.2. Metrics

4.2.1. Size

Lines of Code (LOC)	
Why?	As an indicator of the work being accomplished and for the calculation of other metrics (for example defect density).
How?	It should be agreed whether to count all the lines, or omit the empty ones and comments, which files to exclude, and so on.
When?	From the start of the implementation until stabilization.
Who?	Automated tool.
For whom?	Project manager.

Table 4.2.1. Lines of Code (LOC)

Test points	
Why?	To track progress. To motivate writing tests and to drive agile behaviors.
How?	One test point is one step in an automatic acceptance test scenario or one line of unit tests.
When?	From the start of the implementation until stabilization.
Who?	Developers of the test suites or an automated tool.
For whom?	Development team, project manager.

Table 4.2.2. Test points

4.2.2. Defects

Cumulative number of defects	
Why?	To track testing effectiveness.
How?	By logging each discovered defect into a defect management system.
When?	Iteration retrospective, release retrospective.
Who?	Defect discoverer.
For whom?	Project manager.

Table 4.2.3. Cumulative number of defects

Number of test suites	
Why?	To track testing effort and compare it against the cumulative number of defects.
How?	By extraction of data from the defect repository.
When?	Iteration retrospective, release retrospective.
Who?	Developers of the test suites or an automated tool.
For whom?	Project manager.

Table 4.2.4. Number of test suites

Defect density (#defects/KLOC)	
Why?	To assess the quality of the software in terms of the lack of defects.
How?	By dividing the cumulative number of defects by 1000*LOC
When?	Iteration retrospective, release retrospective.
Who?	Automated tool.
For whom?	Customers.

Table 4.2.5. Defect density

Defect distribution per origin	
Why?	To decide where to allocate the quality assurance resources.
How?	By recording the root cause of every defect in the defect repository and extracting the data by an automated tool.
When?	Iteration retrospective, release retrospective.
Who?	Defect discoverer (mostly testers) and an automated tool.
For whom?	Development team and project manager (leadership).

Table 4.2.6. Defect distribution per origin

Defect distribution per type	
Why?	To learn what types of defects are the most common and help avoid them in the future.
How?	By recording the type of every defect in the defect repository and extracting the data by an automated tool.
When?	Iteration retrospective, release retrospective.
Who?	Defect discoverer (mostly testers) and an automated tool.
For whom?	Developers.

Table 4.2.7. Defect distribution per type

4.2.3. Requirements and design

Value-to-cost ratio of requirements and design ideas	
Why?	To help prioritize requirements and design ideas. To support customer involvement.
How?	By estimating value and cost, and calculating the ratio.
When?	Iteration planning.
Who?	The customer estimates the value, the developers estimate the cost.
For whom?	Development team and project manager (leadership).

Table 4.2.8. Value-to-cost ratio of requirements and design ideas

4.2.4. Quality indicators

Chidamber and Kemerer suite	
Why?	To assess the quality of object-oriented systems, mainly in terms of maintainability. To guide refactoring and testing.
How?	By following the description of metric in the suite.
When?	Constantly for individuals, at every code check-in or daily build for wider audience. As long as new classes are being added.
Who?	Automated tool.
For whom?	Developers, testers.

Table 4.2.9. Chidamber and Kemerer suite

5. Discussion

The findings described in the previous chapter give an overview of different aspects of the way metrics should be used in agile software development, including the benefits that the affected stakeholders are expected to gain. That constitutes an answer to the research question (see section 1.4). There are, however, several issues that require consideration before drawing the final conclusions.

The currently adopted view in software engineering is that a high quality process will yield high quality products, so in consequence the emphasis in quality assurance has been shifted from product improvement to process improvement [Berki et al., 2007]. Process improvement is seen as a preventive approach as opposed to product improvement, which is corrective [Siakas et al., 2005]. Thus, one could question the usefulness of product metrics. Process improvement, however, can be aided not only with process metrics, but with product metrics as well. Right product metrics can lead to reinforcing good practices, as measurement drives behavior [Hartmann and Dymond, 2006], and they can also be used for high level project management decisions [Atkinson et al., 1998]. Furthermore, the customer's focus is mainly on the product and hardly on the process [Schneidewind, 2011; Umarji and Seaman, 2009], therefore, when evaluating process improvement, it is crucial to measure it in terms of the product quality that is achieved [Schneidewind, 2011]. In conclusion, exclusive focus either on the process or on the product is not sufficient [Satpathy et al., 2000].

Berki et al. [2007] and Fenton and Neil [2000] express the opinion that successful measurement requires a mature development process. Mature processes are well-documented and repeatable, and agile processes do not fully

meet these requirements, especially the latter. In fact, agile methodologies attack the premise that processes are repeatable [Highsmith, 2001; Schwaber and Beedle, 2001] and their emergent nature makes agile software development a learning experience for each project [Mnkandla and Dwolatzky, 2007]. It is also strongly believed that agile methods are orthogonal with best practices of the Capability Maturity Model Integrated (CMMI), which is a well-known process reference framework for the assessment and guidance of organizational improvement toward higher maturity levels [Trujillo et al., 2011]. Does that mean that measurement cannot succeed in agile development? I believe it can. Firstly, because of the people-centric approach, maturity is shifted from the process to its participants – agile methodologies rely on committed and competent professionals [Siakas et al., 2005]. Secondly, merging a process in an organization assessed CMMI level 5 with agile practices has been proven viable [Trujillo et al., 2011]. Finally, agile methodologies are moving toward higher levels of maturity due to a growing body of documentation, the growth in academic research, and the massive exchange of practical experiences among practitioners [Mnkandla and Dwolatzky, 2007].

Considering the assertions on the importance and even necessity of measurement for managing software development (see section 4.1.1) on one hand, and the information on the low utilization of metrics on the other hand (see sections 1.1, 1.2), and that most of the best-known agile methods do not provide true support for project management, nor do they cover all phases of the lifecycle [Abrahamsson et al., 2003], this raises the question as to how it is possible that agile software projects are managed without such an essential tool. Or is measurement not as important as the researchers argue? Perhaps the answer lies in the agile management style combined with the maturity expected from the practitioners. The development team is self-organizing and is

responsible for making the decisions directly concerning their work. The project manager is mainly a facilitator, responsible for organization and coordination of everything that has to be done in the project but not directly related to the actual software production. The leadership role is then divided between the team and the management so that there are no collisions in the competences. Such a cooperation of competent professionals seems to work for agile development. Nevertheless, metrics could still bring an enhancement to that by providing input for making more informed decisions, so as not to rely solely on the individuals' intuition and to put more engineering into software engineering.

Not all of the advocated metrics can be regarded as practical. Business Value Delivered (see section 3.1) appears rather difficult to obtain, and its benefits are uncertain. It is supposed to be a metric toward which all other metrics are aligned, but software professionals normally do not have economic expertise, and therefore may find it hard to take economic factors into consideration while designing or choosing metrics. From the agile point of view, satisfying the customer by providing them with a quality product that they want and need, within the agreed time and budget, should be enough. Whether the product will actually provide business value to the customer is largely out of the software vendor's control and will be certain only after the product has been taken into use. Incidentally, this metric is different than the business value assigned to requirements and designs for their evaluation and prioritization. The latter is an estimation given by the customer in a purely numerical (rather than monetary) form, and expresses importance rather than real economic value.

Another impractical metric are the numerical quality goals for requirements (see section 3.4). The metric's author himself admits that quality

characteristics are hard to quantify, and the need to do so is questionable. Far more important is the understanding of these requirements and the reasons they exist. Following the agile principle of extensive communication and customer involvement facilitates achieving such an understanding.

In the literature used for this thesis, the least considered stakeholder is the end user. That may be because usually the user is not directly involved in the development process. However, the scarcity of user-oriented product metrics is puzzling. If they are mentioned, then rather briefly and without any emphasis on their importance. Of course, striving for making a quality product will support the characteristics important to the user. Nonetheless, paying attention only to the customer may not be the optimal approach, as there will likely be a gap between the user's needs and the customer's understanding of those needs. With all the consideration given to customer satisfaction in the agile paradigm and emphasis on the control of the benefits delivered to the customer in the metrics literature, an explicit mention of the end user's satisfaction and benefits is a direction which should be explored in the future research on agile metrics.

6. Conclusions

6.1. Summary

Agile methodologies could benefit from incorporating software product metrics, and the incorporation is possible without compromising agile values and principles. Some of the possible benefits are similar to those in traditional software development, namely early detection of problems and defects, which leads to cost savings, improved understanding of the strengths and weaknesses of the process and product, and support for decision-making, planning, controlling, and monitoring of project plans. Especially important in the agile context are control of the benefits delivered to the customer, motivation for and reinforcement of desired behaviors, assessment of the effects of the adopted practices on product quality, and providing directions on quality assurance activities (such as refactoring and testing) so that neither too much nor too little time is spent on them.

The most important tangible ingredient of agile measurement, especially product measurement, are automated tools, which ensure minimum effort required for metrics collection, interpretation, visualization, and reporting. What tools cannot provide, however, is true understanding of the meaning of a metric and its results. For that, communication – also crucial for agile development otherwise – is necessary. Feedback must be rapid and acted upon quickly. In order for metrics to provide value to the organization, the results of measurement have to be used for improvement, and therefore it must be assured that the purpose of each metric is understood and there is a plan for situations when the actual results deviate from the expected ones. Trust has to be maintained at all times by keeping metrics of individuals confidential and avoiding intrusive supervision, for example publishing product quality metrics

extracted only from the collective source code repository, leaving those from developers' workstations.

A suitable method for choosing what to measure is Goal-Question-Metric (GQM). The emergent nature of agile processes implies that the whole metric set does not have to be designed up front, metrics should rather be let to emerge as the method of working emerges. The metrics that have been validated and appear the most promising for agile development are: the Chidamber and Kemerer suite, size in terms of test points and LOC, number of defects, defect density, defect distribution per origin and per type, testing effort as the number of test suites, value-to-cost ratio for requirements and design ideas.

Measurement should be present throughout the whole development lifecycle. Automated data extraction from the source code repository should be executed regularly an often – whenever new code is submitted, or during the daily build procedure. Local measurement on the workstation of each developer can be ran constantly in order to guide the coding, but the results should not be published, so as not to create an atmosphere of constant supervision. Iteration and release retrospectives are the times when the long-term results of metrics can be analyzed and used as input for improvement decisions. For each metric it should be planned how long it will be used. Measurement should be continued as long as it provides value, and some metrics lose their effectiveness with time.

Automated tools should perform most, if not all, measurements. In case some manual work is needed in that regard, it should be done by the person who needs the data in question. Reporting will normally be the project manager's job.

Different stakeholders have different interests in software and different perceptions of quality attributes, therefore they focus on different metrics. It is

crucial to be aware of the purpose of a metric, because only then one can know who can benefit from it. The interests of some stakeholders overlap partially, as the customer wants the product to have qualities that will be attractive to the user, the project manager has to ensure customer satisfaction while also supporting the development team, and the development team is obliged to provide the best possible solutions, not the easiest ones. The stakeholders' own interests are: good value for money for the user, high business value (return on investment) for the customer, decision-making support for the project manager, and achieving optimal results with the lowest possible effort for the development team. Product metrics described in this thesis provide the most direct benefits to the development team and the project manager, as no explicit user-oriented metrics were found in the literature, and business value has been found difficult and impractical to measure. The user's benefit may be a high quality product for a reasonable price, as product metrics help both ensure quality and reduce the development costs. The customer's benefit is the insight into the product quality and reduced price of the development. The development team and the project manager benefit from the support for decision-making and product quality management.

6.2. Limitations

This research was based only on literature, therefore it is purely theoretical – the results have not been validated empirically. The study was aimed mainly at industrial software development, where a customer (usually a paying one) is present. Therefore, the applicability to open-source development is limited.

The presented results are not a holistic solution, because using product metrics alone is not enough, it has to be complemented with process metrics. Another vast issue not included in the scope is the introduction of a measurement program to an organization.

6.3. Future work

Future work directly resulting from this thesis would be an empirical validation of the results. One metric that has not been researched much and appears particularly suitable for agile development is size measurement by test points.

Another closely related subject lacking in the existing literature are user-oriented product metrics for agile software development, in both theoretical and practical aspects.

An important issue that needs to be resolved is the applicability of the ISO 9126 quality standard to agile software development, as the existing research contains discrepancies in this regard.

References

[Abrahamsson et al., 2002] Pekka Abrahamsson, Outi Salo, Jussi Ronkainen, and Juhani Warsta, *Agile software development methods*, VTT Publications, 2002.

[Abrahamsson et al., 2003] Pekka Abrahamsson, Juhani Warsta, Mikko T. Siponen, and Jussi Ronkainen, New directions on agile methods: a comparative analysis. In: *Proceedings of the 25th International Conference on Software Engineering*, IEEE Computer Society, 2003, 244-254.

[Ambler, 2005] Scott W. Ambler, Measure me wisely. *Software Development* **13** (7), 2005, 59-61. Also available as *http://drdobbs.com/184415360*

[Ambu et al., 2006] Walter Ambu, Giulio Concas, Michele Marchesi, and Sandro Pinna, Studying the Evolution of Quality Metrics in an Agile/Distributed Project. In: Pekka Abrahamsson, Michele Marchesi, and Giancarlo Succi (eds.), *Extreme Programming and Agile Processes in Software Engineering*, Springer Berlin / Heidelberg, 2006, 85-93.

[Bartels et al., 2009] Rodrigo A. Bartels, Jose Rodriguez, and Marcelo Jenkins, Implementing software metrics in an agile organization: a case study. In: *COMPEDES09: II Congreso Computación para el Desarrollo*, 2009.

[Basili, 1992] Victor R. Basili, *Software Modeling And Measurement: The Goal/Question/Metric Paradigm*, University of Maryland at College Park, 1992.

[Basili and Weiss, 1984] Victor R. Basili and David M. Weiss, A methodology for collecting valid software engineering data. *IEEE Transactions on Software Engineering* **10** (6), 1984, 728.

[Basili et al., 1994] Victor R. Basili, Gianluigi Caldiera, and H. D. Rombach, The Goal Question Metric Approach. In: John J. Marciniak (ed.), *Encyclopedia of Software Engineering*, John Wiley & Sons, 1994, 528-532.

[Basili et al., 1996] Victor R. Basili, Lionel C. Briand, and Walcelio L. Melo, A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on Software Engineering* **22** (10), 1996, 751.

[Beck, 1999] Kent Beck, Embracing change with extreme programming. *Computer* **32** (10), 1999, 70-77.

[Beck, 2000] Kent Beck, *Extreme Programming Explained: Embrace Change*, Addison-Wesley Longman Publishing Co., Inc, 2000.

[Beck et al., 2001] Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, and Dave Thomas, Manifesto for Agile Software Development, 2001. *http://www.agilemanifesto.org/*, last accessed January 2012

[Beizer, 1990] Boris Beizer, *Software Testing Techniques (2nd ed.)*, Van Nostrand Reinhold Co, 1990.

[Berki et al., 2007] Eleni Berki, Kerstin Siakas, and Elli Georgiadou, Agile Quality or Depth of Reasoning? Applicability vs. Suitability with Respect to Stakeholders' Needs. In: Ioannis G. Stamelos and Panagiotis Sfetsos (eds.), *Agile Software Development Quality Assurance*, Information Science Reference, 2007, 23-55.

[Bloch, 2001] Joshua Bloch, *Effective Java Programming Language Guide*, Sun Microsystems, Inc, 2001.

[Bøegh, 2006] Jørgen Bøegh, Certifying software component attributes. *IEEE Software* **23** (3), 2006, 74-81.

[Boehm, 1981] Barry W. Boehm, *Software Engineering Economics*, Prentice Hall, 1981.

[Boehm et al., 1976] Barry W. Boehm, John R. Brown, and M. Lipow, Quantitative evaluation of software quality. In: *Proceedings of the 2nd international conference on Software engineering*, IEEE Computer Society Press, 1976, 592-605.

[Bryton and Brito e Abreu, 2009] Sérgio Bryton and Fernando Brito e Abreu, Strengthening refactoring: towards software evolution with quantitative and experimental grounds. In: *Proceedings of the 2009 Fourth International Conference on Software Engineering Advances*, IEEE Computer Society, 2009, 570-575.

[Bundschuh and Dekkers, 2008] Manfred Bundschuh and Carol Dekkers, Software Measurement and Metrics: Fundamentals. In: *The IT Measurement Compendium*, Springer-Verlag Berlin Heidelberg, 2008, 179-206.

[Chidamber and Kemerer, 1994] Shyam R. Chidamber and Chris F. Kemerer, A metrics suite for object oriented design. *IEEE Transactions on Software Engineering* **20** (6), 1994, 476-493.

[Chidamber et al., 1998] Shyam R. Chidamber, David P. Darcy, and Chris F. Kemerer, Managerial use of metrics for object-oriented software: an exploratory analysis. *IEEE Transactions on Software Engineering* **24** (8), 1998, 629-639.

[Cockburn, 2002] Alistair Cockburn, *Agile Software Development*, Addison-Wesley Longman Publishing Co., Inc, 2002.

[Concas et al., 2008] Giulio Concas, Marco Francesco, Michele Marchesi, Roberta Quaresima, and Sandro Pinna, Study of the evolution of an agile project featuring a web application using software metrics.

In: *PROFES '08: Proceedings of the 9th international conference on Product-Focused Software Process Improvement*, Springer-Verlag, 2008, 386-399.

[Daskalantonakis, 1992] Michael K. Daskalantonakis, A practical view of software measurement and implementation experiences within motorola. *IEEE Transactions on Software Engineering* **18** (11), 1992, 998-1010.

[DeMarco, 1982] Tom DeMarco, *Controlling Software Projects: Management, Measurement, And Estimation*, Yourdon Press, New York, NY, 1982.

[Dubinsky et al., 2005] Yael Dubinsky, David Talby, Orit Hazzan, and Arie Keren, Agile metrics at the Israeli Air Force. In: *Proceedings of the Agile Development Conference*, IEEE Computer Society, 2005, 12-19.

[Dybå and Dingsøyr, 2008] Tore Dybå and Torgeir Dingsøyr, Empirical studies of agile software development: A systematic review. *Information and Software Technology* **50** (9-10), 2008, 833-859.

[Fenton and Neil, 2000] Norman E. Fenton and Martin Neil, Software metrics: roadmap. In: *ICSE '00: Proceedings of the Conference on The Future of Software Engineering*, ACM, 2000, 357-370.

[Fenton and Pfleeger, 1998] Norman E. Fenton and Shari L. Pfleeger, *Software Metrics: A Rigorous and Practical Approach*, PWS Publishing Co., 1998.

[Georgiadou, 2003] Elli Georgiadou, Software process and product improvement: a historical perspective. *Cybernetics and Systems Analysis* **39** (1), 2003, 125-142.

[Georgiadou et al., 2003] Elli Georgiadou, Kerstin V. Siakas, and Eleni Berki, Quality improvement through the identification of controllable and uncontrollable factors in software development. In: *EuroSPI 2003 proceedings: European software process improvement*, Technischen Universität, Graz, Austria, 2003, 31-45.

[Gilb and Brodie, 2007] Tom Gilb and Lindsey Brodie, What's Wrong with Agile Methods? Some Principles and Values to Encourage Quantification. In: Ioannis G. Stamelos and Panagiotis Sfetsos (eds.), *Agile Software Development Quality Assurance*, Information Science Reference, 2007, 56-69.

[Gilb and Cockburn, 2008] Tom Gilb and Alistair Cockburn, Point/Counterpoint. *IEEE Software* **25** (2), 2008, 64-67.

[Gopal et al., 2002] Anandasivam Gopal, M. S. Krishnan, Tridas Mukhopadhyay, and Dennis R. Goldenson, Measurement programs in software development: determinants of success. *IEEE Transactions on Software Engineering* **28** (9), 2002, 863-875.

[Hall and Fenton, 1997] Tracy Hall and Norman Fenton, Implementing effective software metrics programs. *IEEE Software* **14** (2), 1997, 55-65.

[Harjumaa et al., 2008] Lasse Harjumaa, Jouni Markkula, and Markku Oivo, How does a measurement programme evolve in software organizations? In: *PROFES '08: Proceedings of the 9th international conference on Product-Focused Software Process Improvement*, Springer-Verlag, 2008, 230-243.

[Hartmann and Dymond, 2006] Deborah Hartmann and Robin Dymond, Appropriate agile measurement: using metrics and diagnostics to deliver business value. In: *AGILE '06: Proceedings of the conference on AGILE 2006*, IEEE Computer Society, 2006, 126-134.

[Heidenberg and Porres, 2010] Jeanette Heidenberg and Ivan Porres, Metrics functions for kanban guards. In: *Proceedings of the 2010 17th IEEE International Conference and Workshops on the Engineering of Computer-Based Systems*, IEEE Computer Society, 2010, 306-310.

[Henderson-Sellers, 1996] Brian Henderson-Sellers, *Object-oriented Metrics: Measures Of Complexity*, Prentice-Hall, Inc, 1996.

[Highsmith, 2000] Jim Highsmith, *Adaptive Software Development: A collaborative Approach To Managing Complex Systems*, Dorset House Publishing Co., Inc, 2000.

[Highsmith, 2001] Jim Highsmith, The great methodologies debate: Part 1: Today, a new debate rages: Agile software development vs. rigorous software development. *Cutter IT Journal* **14** (12), 2001, 2-4.

[Hwong et al., 2007] Beatrice M. Hwong, Gilberto Matos, Monica McKenna, Christopher Nelson, Gergana Nikolova, Arnold Rudorfer, Xiping Song, Grace Y. Tai, Rajanikanth Tanikella, and Bradley Wehrwein, Quality Improvements from using Agile Development Methods: Lessons Learned. In: Ioannis G. Stamelos and Panagiotis Sfetsos (eds.), *Agile Software Development Quality Assurance*, Information Science Reference, 2007, 221-235.

[ISO/IEC, 2001] International Organization for Standardization/International Electrotechnical Commission, ISO/IEC 9126. Software engineering – Product quality, ISO/IEC, 2001.

[Iversen and Mathiassen, 2000] Jakob Iversen and Lars Mathiassen, Lessons from implementing a software metrics program. In: *Proceedings of the 33rd Hawaii International Conference on System Sciences – Volume 7*, IEEE Computer Society, 2000, 7040.

[Jeffries, 2004] Ron Jeffries, A metric leading to agility, 2004. *http://xprogramming.com/xpmag/jatRtsMetric*, last accessed December 2011

[Johnson et al., 2005] Philip M. Johnson, Hongbing Kou, Michael Paulding, Qin Zhang, Aaron Kagawa, and Takuya Yamashita, Improving software development management through software project telemetry. *IEEE Software* **22** (4), 2005, 76-85.

[Kaner and Bond, 2004] Cem Kaner and Walter P. Bond, Software engineering metrics: what do they measure and how do we know? In: *10th International Software Metrics Symposium (METRICS 2004)*, IEEE Computer Society, 2004.

[Kile and Inampudi, 2007] James F. Kile and Maheshwar R. Inampudi, Agile Software Development Quality Assurance: Agile Project Management, Quality Metrics, and Methodologies. In: Ioannis G. Stamelos and Panagiotis Sfetsos (eds.), *Agile Software Development Quality Assurance*, Information Science Reference, 2007, 187-205.

[Kitchenham, 2010] Barbara Kitchenham, What's up with software metrics? – A preliminary mapping study. *Journal of Systems and Software* **83** (1), 2010, 37-51.

[Kogure and Akao, 1983] Masao Kogure and Yoji Akao, Quality Function Deployment and CWQC in Japan. *Quality Progress* **16** (10), 1983, 25-29.

[Kruchten, 2000] Philippe Kruchten, *The Rational Unified Process: An Introduction*, Addison-Wesley Longman Publishing Co., Inc, 2000.

[Kruchten, 1996] Philippe Kruchten, A rational development process. *Crosstalk*, 1996, 11-16.

[Ktata and Lévesque, 2010] Oualid Ktata and Ghislain Lévesque, Designing and implementing a measurement program for Scrum teams: what do agile developers really need and want? In: *Proceedings of the Third C* Conference on Computer Science and Software Engineering*, ACM, 2010, 101-107.

[Kunz et al., 2008] Martin Kunz, Reiner R. Dumke, and Niko Zenker, Software metrics for agile software development. In: *ASWEC '08: Proceedings of the 19th Australian Conference on Software Engineering*, IEEE Computer Society, 2008, 673-678.

[Layman et al., 2004] Lucas Layman, Laurie Williams, and Lynn Cunningham, Motivations and measurements in an agile case study. In: *Proceedings of the 2004 workshop on Quantitative techniques for software agile process*, ACM, 2004, 14-24.

[Leffingwell, 2006] Dean Leffingwell, Iteration and release retrospectives: the natural rhythm for agile measurement, 2006. *http://www.agilejournal.com/articles/columns/column-articles/51-iteration-and-release-retrospectives-the-natural-rhythm-for-agile-measurement*, last accessed December 2011

[Li, 1999] Wei Li, Software product metrics. Using them to quantify design and code quality. *IEEE Potentials* **18** (5), 1999, 24-27.

[Li and Henry, 1993] Wei Li and Sallie Henry, Object-oriented metrics that predict maintainability. *Journal of Systems and Software* **23** (2), 1993, 111.

[Lincke and Löwe, 2007] Rüdiger Lincke and Welf Löwe, Compendium of Software Quality Standards and Metrics, 2007. *http://www.arisa.se/compendium/*, last accessed January 2012

[Lindvall et al., 2002] Mikael Lindvall, Victor R. Basili, Barry W. Boehm, Patricia Costa, Kathleen Dangle, Forrest Shull, Roseanne Tesoriero, Laurie A. Williams, and Marvin V. Zelkowitz, Empirical findings in agile methods. In: *Proceedings of the Second XP Universe and First Agile Universe Conference on Extreme Programming and Agile Methods – XP/Agile Universe 2002*, Springer-Verlag, 2002, 197-207.

[Martin, 2003] Robert C. Martin, *Agile Software Development: Principles, Patterns, and Practices*, Prentice Hall PTR, 2003.

[McCabe, 1976] Thomas J. McCabe, A complexity measure. In: *Proceedings of the 2nd international conference on Software engineering*, IEEE Computer Society Press, 1976, 407.

[McCall et al., 1977] Jim A. McCall, Paul K. Richards, and Gene F. Walters, *Factors in software quality: concept and definitions of software quality*, General Electric, Rome Air Development Center, 1977.

[Mnkandla and Dwolatzky, 2007] Ernest Mnkandla and Barry Dwolatzky, Agile Software Methods: State-of-the-Art. In: Ioannis G. Stamelos and Panagiotis Sfetsos (eds.), *Agile Software Development Quality Assurance*, Information Science Reference, 2007, 2-22.

[Niessink and van Vliet, 1999] Frank Niessink and Hans van Vliet, Measurements should generate value, rather than data. In: *Proceedings of the 6th International Symposium on Software Metrics*, IEEE Computer Society, 1999, 31.

[Offen and Jeffery, 1997] Raymond J. Offen and Ross Jeffery, Establishing software measurement programs. *IEEE Software* **14** (2), 1997, 45-53.

[Olague et al., 2007] Hector M. Olague, Letha H. Etzkorn, Sampson Gholston, and Stephen Quattlebaum, Empirical validation of three software metrics suites to predict fault-proneness of object-oriented classes developed using highly iterative or agile software development processes. *IEEE Transactions on Software Engineering* **33** (6), 2007, 402-419.

[Ordonez and Haddad, 2008] Mauricio J. Ordonez and Hisham M. Haddad, The state of metrics in software industry. In: *ITNG '08: Proceedings of the Fifth International Conference on Information Technology: New Generations*, IEEE Computer Society, 2008, 453-458.

[Palmer and Felsing, 2001] Steve R. Palmer and Mac Felsing, *A Practical Guide to Feature-Driven Development*, Pearson Education, 2001.

[Paulish and Carleton, 1994] Daniel J. Paulish and Anita D. Carleton, Case studies of software-process-improvement measurement. *Computer* **27** (9), 1994, 50-57.

[Pfleeger, 1993] Shari L. Pfleeger, Lessons learned in building a corporate metrics program. *IEEE Software* **10** (3), 1993, 67-74.

[Salo et al., 2002] Outi Salo, Maarit Tihinen, and Matias Vierimaa, Enabling comprehensive use of metrics. In: *Proceedings of the 4th International Conference on Product Focused Software Process Improvement*, Springer-Verlag, 2002, 326-336.

[Satpathy et al., 2000] Manoranjan Satpathy, Rachel Harrison, Colin Snook, and Michael J. Butler, A generic model for assessing process quality. In: *Proceedings of the 10th International Workshop on New Approaches in Software Measurement*, Springer-Verlag, 2000, 94-110.

[Schneidewind, 2011] Norman Schneidewind, What can software engineers learn from manufacturing to improve software process and product? *Journal of Intelligent Manufacturing* **22** (4), 2011, 597-606.

[Schwaber, 1995] Ken Schwaber, SCRUM development process. In: *Proceedings of the 10th Annual ACM Conference on Object Oriented Programming Systems, Languages, and Applications (OOPSLA)*, 1995, 117-134.

[Schwaber and Beedle, 2001] Ken Schwaber and Mike Beedle, *Agile Software Development with Scrum*, Prentice Hall PTR, 2001.

[Scotto et al., 2006] Marco Scotto, Alberto Sillitti, Giancarlo Succi, and Tullio Vernazza, A non-invasive approach to product metrics collection. *Journal of Systems Architecture* **52** (11), 2006, 668-675.

[Sfetsos and Stamelos, 2010] Panagiotis Sfetsos and Ioannis Stamelos, Empirical studies on quality in agile practices: a systematic literature review. In: *Proceedings of the 2010 Seventh International Conference on the Quality of Information and Communications Technology*, IEEE Computer Society, 2010, 44-53.

[Siakas et al., 1997] Kerstin Siakas, Eleni Berki, Elli Georgiadou, and Chris Sadler, The complete alphabet of quality software systems: conflicts and compromises. In: *7th World Congress on Total Quality & Qualex 97*, 1997, 603-618.

[Siakas et al., 2005] Kerstin V. Siakas, Elli Georgiadou, and Eleni Berki, Agile methodologies and software process improvement. In: *Proceedings of the Virtual Multi Conference on Computer Science and Information Systems*, International Association for the Development of Information Society (IADIS), 2005, 412-417.

[Stapleton, 1997] Jennifer Stapleton, *Dynamic Systems Development Method*, Addison Wesley, 1997.

[Stark et al., 1994] George Stark, Robert C. Durst, and C. W. Vowell, Using metrics in management decision making. Computer **27** (9), 1994, 42-48.

[Suryn et al., 2002] Witold Suryn, Pierre Bourque, Alain Abran, and Claude Laporte, Software product quality practices quality measurement and evaluation using TL9000 and ISO/IEC 9126. In: *Proceedings of the 10th International Workshop on Software Technology and Engineering Practice*, IEEE Computer Society, 2002, 156.

[Talby et al., 2005] David Talby, Ori Nakar, Noam Shmueli, Eli Margolin, and Arie Keren, A process-complete automatic acceptance testing framework. In: *Proceedings of the IEEE International Conference on Software – Science, Technology and Engineering*, IEEE Computer Society, 2005, 129-138.

[Trujillo et al., 2011] Miguel M. Trujillo, Hanna Oktaba, Francisco J. Pino, and María J. Orozco, Applying agile and lean practices in a software development project into a CMMI organization. In: *Proceedings of the 12th international conference on Product-focused software process improvement*, Springer-Verlag, 2011, 17-29.

[Umarji and Seaman, 2009] Medha Umarji and Carolyn Seaman, Gauging acceptance of software metrics: Comparing perspectives of managers and developers. In: *ESEM '09: Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement*, IEEE Computer Society, 2009, 236-247.

[UnitMetrics, 2007] UnitMetrics, 2007. *http://unitmetrics.sourceforge.net/*, last accessed in January 2012.

[Van Schooenderwoert, 2006] Nancy Van Schooenderwoert, Embedded agile project by the numbers with newbies. In: *AGILE '06: Proceedings of the conference on AGILE 2006*, IEEE Computer Society, 2006, 351-366.

[Yang et al., 2008] Ye Yang, Mei He, Mingshu Li, Qing Wang, and Barry Boehm, Phase distribution of software development effort. In: *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement*, ACM, 2008, 61-69.

Appendix A

This appendix contains descriptions of the metrics found in Figure 2.1.1 (page 9) based on the Compendium of Software Quality Standards and Metrics [Lincke and Löwe, 2007].The abbreviations and full names of the metrics are written in bold face. The order of the metrics is the same as in the figure.

LOC – Lines of Code simply counts the lines of source code (line break characters) of a certain software entity. It is a simple yet powerful metric to assess the complexity of software entities. Since it is depending on code conventions and format, it is critical to use it in generated codes since it may lack of line breaks. Additionally it can only be measured in the source code itself from the front-end and is therefore a front-end side metric.

SIZE2 – Number of Attributes and Methods simply counts the number of attributes and methods of a class. It is an object-oriented metric that can be applied to modular languages by considering the number of variables (globally visible in a module) and its number of functions and procedures.

NOM – Number of local Methods measures the number of methods locally declared in a class. Inherited methods are not considered. It is the size of the interface of a class and allows conclusions on its complexity.

CC – McCabe Cyclomatic Complexity is a measure of the control structure complexity of software. It is the number of linearly independent paths and therefore, the minimum number of independent paths when executing the software.

WMC – Weighted Method Count is a weighted sum of methods implemented within a class. It is parameterized by a way to compute the weight of each method. Possible weight metrics are:

- McCabe Cyclomatic Complexity,
- Lines of Code,
- 1 (unweighted WMC).

RFC – Response For a Class is a count of (public) methods in a class and methods directly called by these. RFC is only applicable to object-oriented systems.

DIT – Depth of Inheritance Tree is the maximum length of a path from a class to a root class in the inheritance structure of a system. DIT measures how many super-classes can affect a class. DIT is only applicable to object-oriented systems.

NOC – Number of Children is the number of immediate subclasses (children) subordinated to a class (parent) in the class hierarchy. NOC measures how many classes inherit directly methods or fields from a super-class. NOC is only applicable to object-oriented systems.

Ca – Afferent Coupling between packages measures the total number of external classes coupled to classes of a package due to incoming coupling (coupling from classes external classes of the package, uses CBO definition of coupling). Each class counts only once. Zero if the package does not contain any classes or if external classes do not use the package's classes. Ca is primarily applicable to object-oriented systems.

CBO – Coupling Between Objects is the number of other classes that a class is coupled to. CBO is only applicable to object-oriented systems.

CDBC – Change Dependency Between Classes measures the class level coupling. It is a measure assigned to pairs of classes describing how

dependent one class (client class) is on the other (server class). This allows conclusions on the follow-up work to be done in a client class, when the server class is changed in the course of re-engineering.

CDOC – Change Dependency of Classes measures the class level coupling. It is a measure assigned to classes describing how dependent other classes (client classes) are on this class (server class). This allows conclusions on the follow-up work to be done in all client class, when the server class is changed in the course of re-engineering. It is a accumulation of the CDBC metric, for a server class (SC) and all its client classes (CC).

Ce – Efferent Coupling between packages measures the total number of external classes coupled to classes of a package due to outgoing coupling (coupling to classes external classes of the package, uses Ce definition of coupling). Each class counts only once. Zero if the package does not contain any classes or if external classes are not used by the package's classes. Ce is primarily applicable to object-oriented systems.

CF – Coupling Factor measures the coupling between classes excluding coupling due to inheritance. It is the ratio between the number of actually coupled pairs of classes in a scope (e.g., package) and the possible number of coupled pairs of classes. CF is primarily applicable to object-oriented systems.

DAC – Data Abstraction Coupling measures the coupling complexity caused by Abstract Data Types (ADTs). This metric is concerned with the coupling between classes representing a major aspect of the object oriented design, since the reuse degree, the maintenance and testing effort for a class are decisively influenced by the coupling level between classes. Basically same as DAC, but coupling limited to type references.

I – Instability between packages measures the ratio between the outgoing and the total number of in- and outgoing couplings from classes inside the package from/to classes outside the package (coupling to classes external classes of the package, uses I definition of coupling). Each class counts only once. Zero if the package does not contain any classes or if external classes are not used by the package's classes. I is primarily applicable to object-oriented systems.

LD – Locality of Data metric relates the amount of data being local the class to the total amount of data used by the class. This relates to the quality of abstraction embodied by the class and allows conclusions on the reuse potential of the class and testability.

MPC – Message Passing Coupling measures the number of method calls defined in methods of a class to methods in other classes, and therefore the dependency of local methods to methods implemented by other classes. It allows for conclusions on the message passing (method calls) between objects of the involved classes. This allows for conclusions on re-useability, maintenance and testing effort.

PDAC – Package Data Abstraction Coupling** measures the coupling complexity caused by ADTs on package level. Based on DAC it transfers the effects of the coupling between classes on the reuse degree, maintenance and testing effort to more abstract organization units like packages and modules, which are as well decisively influenced by the coupling between classes of different packages.

LCOM – Lack of Cohesion in Methods metric is a measure for the number of not connected method pairs in a class representing independent parts having no cohesion. It represents the difference between the number of

method pairs not having instance variables in common, and the number of method pairs having common instance variables.

ILCOM – Improvement of LCOM is a measure for the number of connected components in a class. A component are methods of a class sharing (being connected by) instance variables of the class. The less separate components there are the higher is the cohesion of the methods in the class.

TCC – Tight Class Cohesion metric measures the cohesion between the public methods of a class. That is the relative number of directly connected public methods in the class. Classes having a low cohesion indicate errors in the design.

LOD – Lack of Documentation means how many comments are lacking in a class, considering one class comment and a comment per method as optimum. Structure and content of the comments are ignored.

Appendix B

This appendix contains descriptions of the quality characteristics (properties) and their sub-characteristics found in Figure 2.1.1 (page 9) based on the Compendium of Software Quality Standards and Metrics [Lincke and Löwe, 2007].The names of the main characteristics are written in bold face and italicized, and the names of their sub-characteristics are only italicized. Each main characteristic is followed by its sub-characteristic, in the same order as in the figure.

The ***functionality*** characteristic allows to draw conclusions about how well software provides desired functions. It can be used for assessing, controlling and predicting the extent to which the software product (or parts of it) in question satisfies functional requirements.

The *suitability* sub-characteristic allows to draw conclusions about how suitable software is for a particular purpose. It correlates with metrics which measure attributes of software that allow to conclude the presence and appropriateness of a set of functions for specified tasks.

The *accuracy* sub-characteristic allows to draw conclusions about how well software achieves correct or agreeable results. It correlates with metrics which measure attributes of software that allow to conclude about its provision of correct or agreeable results.

The *interoperability* sub-characteristic allows to draw conclusions about how well software interacts with designated systems. It correlates with metrics which measure attributes of software that allow to conclude about its ability to interact with specified systems.

The *security* sub-characteristic allows to draw conclusions about how secure software is. It correlates with metrics which measure attributes of

software that allow to conclude about its ability to prevent unauthorized access, whether accidental or deliberate, to programs or data.

The *functionality compliance* sub-characteristic allows to draw conclusions about how well software adheres to application related standards, conventions, and regulations in laws and similar prescriptions. It correlates with metrics which measure attributes of software that allow to conclude about the adherence to application related standards, conventions, and regulations in laws and similar prescriptions.

The ***reliability*** characteristic allows to draw conclusions about how well software maintains the level of system performance when used under specified conditions. It can be used for assessing, controlling and predicting the extent to which the software product (or parts of it) in question satisfies reliability requirements.

The *maturity* sub-characteristic allows to draw conclusions about how mature software is. It correlates with metrics which measure attributes of software that allow to conclude about the frequency of failure by faults in the software.

The *fault-tolerance* sub-characteristic allows to draw conclusions about how fault-tolerant software is. It correlates with metrics which measure attributes of software that allow to conclude on its ability to maintain a specified level of performance in case of software faults or infringement of its specified interface.

The *recoverability* sub-characteristic allows to draw conclusions about how well software recovers from software faults or infringement of its specified interface. It correlates with metrics which measure attributes of software that allow to conclude on its ability to re-establish its level of performance and recover the data directly affected in case of a failure.

The *reliability compliance* sub-characteristic allows to draw conclusions about how well software adheres to application related standards, conventions, and regulations in laws and similar prescriptions. It correlates with metrics which measure attributes of software that allow to conclude about the adherence to application related standards, conventions, and regulations in laws and similar prescriptions.

The ***usability*** characteristic allows to draw conclusions about how well software can be understood, learned, used and liked by the developer. It can be used for assessing, controlling and predicting the extent to which the software product (or parts of it) in question satisfies usability requirements.

The *understandability* sub-characteristic allows to draw conclusions about how well users can recognize the logical concepts and applicability of software. It correlates with metrics which measure attributes of software that allow to conclude on the users' efforts for recognizing the logical concepts and applicability. Users should be able to select a software product which is suitable for their intended use.

The *learnability* sub-characteristic allows to draw conclusions about how well users can learn the application of software. It correlates with metrics which measure attributes of software that allow to conclude on the users' efforts for learning the application of software.

The *operability* sub-characteristic allows to draw conclusions about how well users can operate software. It correlates with metrics which measure attributes of software that allow to conclude on the users' efforts for operation and operation control.

The *attractiveness* sub-characteristic allows to draw conclusions about how attractive software is to the user. It correlates with metrics which measure

attributes of software that allow to conclude on the capability of the software product to be attractive to the user.

The *usability compliance* sub-characteristic allows to draw conclusions about how well software adheres to application related standards, conventions, and regulations in laws and similar prescriptions. It correlates with metrics which measure attributes of software that allow to conclude about the adherence to application related standards, conventions, and regulations in laws and similar prescriptions.

Re-Usability is a special case of the quality factor Usability. It can be understood as usability of software for integration in other systems, e.g., usability of libraries and components. Here the user is a software engineer/developer. Hence, internal usability metrics are used for predicting the extent of which the software in question can be understood, learned, operated, is attractive and compliant with usability regulations and guidelines where here using means integrating it in a larger software system.

Understandability for Reuse is a special case of the quality criterion Understandability. It can be understood as understandability of software for a software engineer/developer with the purpose of integrating it in new systems, e.g., understandability of libraries and components. Hence, software engineers/developers should be able to select a software product which is suitable for their intended use. Internal understandability reuse metrics assess whether new software engineers/developers can understand: whether the software is suitable; how it can be used for particular tasks.

Learnability for Reuse is a special case of the quality criterion Learnability. It can be understood as learnability of software for a software engineer or

developer with the purpose of integrating it in new systems, e.g., learnability of libraries and components. Hence, internal learnability metrics assess how long software engineers or developers take to learn how to use particular functions, and the effectiveness of documentation. Learnability is strongly related to understandability, and understandability measurements can be indicators of the learnability potential of the software.

Operability for Reuse is a special case of the quality criterion Operability. It can be understood as programmability of software, i.e., the development effort for a software engineer/developer to integrate it in new systems, e.g., programmability software using libraries and components. Hence, internal programmability metrics assess whether software engineers/developers can integrate and control the software.

Attractiveness for Reuse is a special case of the quality criterion Attractiveness. It can be understood as attractiveness of software for a software engineer /developer to integrating it in new systems, e.g., attractiveness of libraries and components. Internal attractiveness metrics assess the appearance of the software. In a reuse context, they will be influenced by factors such as code and documentation layout. In addition to appearance, internal attractiveness metrics for reuse also assess size and complexity of the reused code.

Compliance for Reuse is a special case of the quality criterion Compliance. It can be understood as compliance to re-use standards or conventions or regulations in laws and similar prescriptions. Internal compliance metrics assess adherence to standards, conventions, style guides or regulations relating to re-usability.

The ***efficiency*** characteristic allows to draw conclusions about how well software provides required performance relative to amount of resources used. It can be used for assessing, controlling and predicting the extent to which the software product (or parts of it) in question satisfies efficiency requirements.

The *time behavior* sub-characteristic allows to draw conclusions about how well the time behavior of software is for a particular purpose. It correlates with metrics which measure attributes of software that allow to conclude about the time behavior of the software (or parts of it) in combination with the computer system during testing or operating.

The r*esource utilization* sub-characteristic allows to draw conclusions about the amount of resources utilized by the software. It correlates with metrics which measure attributes of software that allow to conclude about the amount and duration of resources used while performing its function.

The *compliance* sub-characteristic allows to draw conclusions about how well software adheres to application related standards, conventions, and regulations in laws and similar prescriptions. It correlates with metrics which measure attributes of software that allow to conclude about the adherence to application related standards, conventions, and regulations in laws and similar prescriptions.

The ***maintainability*** characteristic allows to draw conclusions about how well software can be maintained. It can be used for assessing, controlling and predicting the effort needed to modify the software product (or parts of it) in question.

The *analyzability* sub-characteristic allows to draw conclusions about how well software can be analyzed. It correlates with metrics which measure attributes of software that allow to conclude about the effort needed for

diagnosis of deficiencies or causes of failures, or for identification of parts to be modified.

The *changeability* sub-characteristic allows to draw conclusions about how well software can be changed. It correlates with metrics which measure attributes of software that allow to conclude about the effort needed for modification, fault removal or for environmental change.

The *stability* sub-characteristic allows to draw conclusions about how stable software is. It correlates with metrics which measure attributes of software that allow to conclude about the risk of unexpected effects as result of modifications.

The *testability* sub-characteristic allows to draw conclusions about how well software can be tested and is tested. It correlates with metrics which measure attributes of software that allow to conclude about the effort needed for validating the software and about the test coverage.

The *maintainability compliance* sub-characteristic allows to draw conclusions about how well software adheres to application related standards, conventions, and regulations in laws and similar prescriptions. It correlates with metrics which measure attributes of software that allow to conclude about the adherence to application related standards, conventions, and regulations in laws and similar prescriptions.

The ***portability*** characteristic allows to draw conclusions about how well software can be ported from one environment to another. It can be used for assessing, controlling and predicting the extent to which the software product (or parts of it) in question satisfies portability requirements.

The *adaptability* sub-characteristic allows to draw conclusions about how well software can be adapted to environmental change. It correlates with metrics which measure attributes of software that allow to conclude

about the amount of changes needed for the adaptation of software to different specified environments.

The *installability* sub-characteristic allows to draw conclusions about how well software can be installed in a designated environment. It correlates with metrics which measure attributes of software that allow to conclude about the effort needed to install the software in a specified environment.

The *co-existence* sub-characteristic allows to draw conclusions about how well software can co-exist with other software products in the same operational environment. It correlates with metrics which measure attributes of software that allow to conclude about the dependencies, concurrency behavior, or side effects.

The *replaceability* sub-characteristic allows to draw conclusions about how well software can replace other software or parts of it. It correlates with metrics which measure attributes of software that allow to conclude about opportunity and effort using it instead of specified other software in the environment of that software.

The *portability compliance* sub-characteristic allows to draw conclusions about how well software adheres to application related standards, conventions, and regulations in laws and similar prescriptions. It correlates with metrics which measure attributes of software that allow to conclude about the adherence to application related standards, conventions, and regulations in laws and similar prescriptions.