Fighting Technical Debt: Enabling Sustainable Productivity

Panu Tunttunen

University of Tampere
School of Information Sciences
Computer Science
M.Sc. Thesis
Supervisor: Timo Poranen
December 2014

University of Tampere
School of Information Sciences
Computer Science
Panu Tunttunen: Fighting Technical Debt: Enabling Sustainable Productivity
M.Sc. Thesis, 74 pages, 6 Appendix pages
December 2014

Abstract

The modern information society is entirely dependable on software. The majority of every day solutions that we use, such as mobile devices, home appliances or vehicles, require software to function. In addition, every business relies on various software systems and services. The reason for software usage is its versatility, which allows business to rapidly adapt to continuous changes, and flexibly revise their objectives and approach new opportunities. The adaptation, competitiveness and survival of organizations are dependable on software that they are using [Boehm, 2006].

Software is purely an intellectual product and considered as one of most labor-intensive, complex, error-prone and economically challenging technology in history. Software is often taken for granted and its existence is not acknowledged until it is malfunctioning [Krasner, 1998].

Without successful software projects there are no working software to serve our needs. To pursue successful software project trajectory, the ability to develop high quality solutions in a profitable velocity is essential. Hence, the factors that enable high and sustainable development productivity require adequate management.

Keywords: software quality, internal quality, technical debt, refactoring, productivity

Acknowledgements

This thesis is dedicated to my loved ones — my family. This is an appreciation for all the support and care that I have received, especially during the difficult phases of my life. Despite all the "technical debt" in our lives, we have to sustain our productivity to keep going forward.

Tampere, December 2014

Table of Contents

1 Introduction

1.1 Background to the Research Topic

The modern information society is entirely dependable on software. The majority of every day solutions that we use, such as mobile devices, home appliances or vehicles, require software to function. In addition, every business relies on various software systems and services. The reason for software usage is its versatility, which allows business to rapidly adapt to continuous changes, and flexibly revise their objectives and approach new opportunities. The adaptation, competitiveness and survival of organizations are dependable on software that they are using [Boehm, 2006].

Software is purely an intellectual product and considered as one of most labor-intensive, complex, error-prone and economically challenging technology in history. Software is often taken for granted and its existence is not acknowledged until it is malfunctioning [Krasner, 1998].

Without successful software projects there are no working software to serve our needs. To pursue successful software project trajectory, the ability to develop high quality solutions in a profitable velocity is essential. Hence, the factors that enable high and sustainable development productivity require adequate management.

1.2 Motivation and Goals for the Research

The culture and literature of "software requirements" systematically fails to acknowledge the majority of critical factors for successful software [Gilb, 2000]. Issues, such as financial budget, delivery deadlines and system performance are often the main concern in software projects. Hence, many relevant factors concerning quality are neglected. The developers of software and their ability to produce good quality source code and architecture come into the picture.

This research discusses internal software quality and its impact on developer productivity, and the phenomenon that causes them to degrade — technical debt. Technical debt means poor quality source code design and architecture that is incurred through development team incompetence or intentional business-driven

decisions. The personal experience concerning technical debt and its extreme consequences leading to project failure has also been a motivation to discuss this topic. The research questions that are inspected are the following:

- Is high internal quality essential in software projects? If so, why?
- How to promote productive development?
- How to sustain productive development?

The research literature concerning internal quality, technical debt and productivity is reviewed and processed. This thesis tries to answer the questions and process these topics by forming an understanding of the essential parts of development productivity.

1.3 Structure of the Thesis

Chapter 2 introduces to agile software development and discusses general challenges and productivity. Internal software quality and its measurement is introduced in Chapter 3. The main issue concerning internal quality and productivity called technical debt is discussed in Chapter 4. Technical debt exists in every software project and affects widely within a project from individual developers to business. Technical debt types, sources and consequences are discussed. Chapter 5 explains refactoring, which is used for source code quality improvement. In Chapter 6, the research questions concerning sustainable development productivity are processed, and in Chapter 7 the thesis is summarized and technical debt is further discussed.

2 Software Development and Productivity

2.1 Introduction to Agile Software Development

Software engineering process consists of definition, implementation, assessment, measurement, management, evolution and improvement of a software life-cycle process [McConnell, 2004]. The purpose is to make the software processes systematic and increase the probability of project success. Solid solutions are needed to cope with the dynamic and challenging software industry.

Software evolution consists of all the changes happening from the initial planning to retirement of a software product [Wagner, 2013]. Today's development practices promote incremental production instead of designing everything upfront. Software source code evolves quite drastically throughout its life-cycle, so any upfront specifications will be modified several times later on. The direction of evolution is wrong, if the software is developed short-term benefits in mind. Software should always be maintainable, so the consequences of short-term decisions have to be dealt with later on.

Continuous changes cause various risks in software development. Agile development methods respond to continuously changing requirements. These methods promote, e.g., working software, adaptive planning, close collaboration and software prototyping. Customer collaboration is emphasized to minimize risks associated with requirements, knowledge and comprehension within a project. The customer is the most important stakeholder of the project, so regular reporting and reviewing is essential. Furthermore, customer representatives can provide useful feedback to the development team to calibrate their understanding and objectives.

Agile procedures are performed in iterations throughout a project. Each iteration consists various development phases, and every iteration is a micro-sized project in a project. Iterations enable adaptive planning and prioritization to address changing user and customer requirements. Iteration usually takes from a week to a month, in which time software is incrementally developed towards the final release. Phases include requirement analysis, planning, software design, software implementation and software testing. Division to iterations enables management

to have better control over schedules and workload. Each iteration is meant to output a working version of the software for the customer to see the progression. Working software acts as continuously evolving prototype for functional testing and is a visual assurance of project progression.

Scrum is an agile methodology, that promotes project roles, such as product owner (PO), Scrum master (SM) and a development team [Schwaber, 2007]. The PO is accountable for project success to the customer, the SM assures correct development activities and the development team designs and develops the software. The source code of the software is collectively owned and maintained in Scrum. As agile methods have iterations, in Scrum they are called sprints. At the end of every sprint, there is a review meeting, in which the Scrum participants go through the progression and planning, and present them to the PO. In addition, a retrospective meeting is held by the SM to ensure improvement in the project process [Shore, 2007]. In Scrum, refactoring is the key practice to maintain high quality source code [Ktata & Lévesque, 2010]. Refactoring means source code modification in a way that improves the readability and understandability without altering its functionality. Refactoring is discussed in more detail in Chapter 5.

Another agile methodology similar to Scrum is Extreme Programming (XP). Performed iterations include, e.g., user stories, iteration planning, development, testing, refactoring and rapid working software versions [Wells, 1999]. XP is heavily based on frequent testing, so test-driven development (TDD) approach is often used [Fowler *et al.*, 1999]. TDD means that software tests are written before the implementations and the implementations are matched against them afterwards. The tests are called "unit tests", because they test only a consistent component in the implemented code. Development velocity is an important metric in XP, because it is used to monitor the progression. In XP, meetings are held as short "stand-up" discussions. Collective ownership and knowledge is promoted, thus collaboration is an important part of XP. Pair programming is promoted, and all production source code is expected to be a result from pair programming.

2.2 Challenges in Development

Software development projects are generally a challenge for the management. A common reason for this is complex software industry specific characteristics, such as human interaction, high complexity and product versatility [Jørgensen, 1999; Subramanian *et al.*, 2007]. Software projects have common problems, such as budget overruns, delivery delays, poor response to user requirements and lack of management [McLeod & MacDonell, 2011].

Software development is all about customization and solutions in a certain context. When every software project is produced uniquely, it is hard to generate work-proof operation plans or estimations. Because projects are unique, the estimations have to be based on estimation models, related data history or estimator expertise. Effort estimation is difficult, because there often are no detailed developer activity information. Furthermore, the effort also depends on the complexity of the software requirements. However, software managers and developers continue making estimations based on their assumptions, although there are various models to be used [Hihn & Habib-agahi, 1991]. Hence, these kind of estimations are made without risk analysis or any data for verification. Estimations based on unreliable assumptions and intuitions are harmful to any project.

Software development projects are considered to have high risks and they also have a reputation for failure [Savolainen *et al.*, 2012]. Projects often fail because the risks involved are not identified and managed. One significant factor to affect the possibility of risks and their consequences is the quality of the project [Ould, 1999]. According to Shihab *et al.* [2012], only 16.2% of software projects are on time and budget, and from the rest 52.7% have reduced functionality and 31.1% are cancelled before completion.

Changes with high risk require additional attention in design, implementation, testing and reviews. Risks can be identified efficiently using different factors, for example, amount of code lines and blocks added by changes, bugs related to the change (bugs in related files and amount of related bug reports) and developer experience. Developers are reported to be accurate 96.1% of the time when identifying changes that might introduce bugs, but are less reliable to detect risky changes [Shihab *et al.*, 2012]. Tomaszewski [2006] states that existing code

modification has considerably bigger risk to introduce problems compared to new code in a new class.

As an example of fragility, source code is a very vulnerable element. Even one undesired character in a source code can crash a whole software system. Majority of solutions are custom made and are not proven to work, so every line of code has to be tested and verified. In architecture, poor design can cause major issues in the future when a software is built on fragile foundation.

Quality and economics are also tightly related fundamental concepts in software development [Wagner, 2013]. However, according to Krasner [1998], there is currently no validated economic theory for software quality. Thus, economic models are rarely used in software industry, because there is lack of common data in the field [Wagner, 2013]. In turn, research sector is lacking general empirical knowledge, since the statistics for software costs are not public data. As a result, forming the required knowledge becomes extremely difficult. Another quality costing issue is the inability to define a stable measurement unit. This is one of the reasons why economic models are not generally recommended for software measurements.

2.3 Development Productivity

Boehm [1981] reports that development productivity is mostly affected by developers and the way they are organized and managed. Mohapatra & Gupta [2011] found that productivity is significantly dependent on technology training. According to Yu *et al.* [1990], the most influential factors in productivity are feature complexity, requirement completeness and stability, developer experience, development environment and tools. Also, architectural decisions are stated to have varying impact on productivity. Developers that have the most knowledge and experience in the application domain are most productive [Sommerville, 2010]. Variations in productivity could be explained by developer experience (measured in KLOC or years) and abstraction level of used programming language [Raza & Faria, 2014].

A global study shows that it takes more than a year for developers to reach full productivity in a software project [Mockus & Weiss, 2001]. Mockus [2009] states

that interactional dimension defines the centrality of a person in the decision-making process. This means that the more experience a developer has, the higher he is in the interactional dimension, thus having more impact in production. Professional programmers do not let good quality source code to degrade [Martin, 2009], hence developers should always deliver good quality source code.

Poor quality process eventually kills productivity, and can make developers to show their frustration and demand improvements to the process. Developers can eventually stop caring if the process is not improved. The outcome might be that concerned employees quit their job and look for more stable working environment to work in. High quitting rate can degrade the public image of the company and affect future employment. The organization might lose valuable knowledge with any leaving employees and regaining an employee with similar knowledge and experience might be expensive. Modern and high quality practices help to keep the skilled developers motivated and attract new talented ones.

Productivity consists of two factors which are the product size and development effort [Tomaszewski, 2006]. The productivity of a developer can be measured in product units produced over certain unit of effort [Mockus, 2009]. In other words, measurement tells how much output does a certain input generate. For example, if full-time developer productivity is measured, the roughly approximated input efforts (months) are multiplied by salary and additional employment costs. When productivity is measured, it has to be inspected in context of overall quality of a software product [IEEE, 1993].

Productivity is usually measured by using function points. Function point analysis (FPA) is functional size measurement of software that is delivered to the customer. Software processes and data are measured by identification, classification and weighting. Function points are calculated from the amount of data manipulated, number of interfaces, amount of user interactions and external inputs and outputs [Sommerville, 2010]. The alternating complexity between functionality features are compensated with a complexity factor, which also is considered as weakness in function point analysis, as it is subjective [Tomaszewski, 2006].

In addition to productivity measurement, the results can be used to indicate software defect ratio, resource assignment or software scope analysis. Function points are considered to be the most popular metric for functionality [Sommerville,

2010]. However, Raza & Faria [2014] list issues with function points, such as measures based on lines of code (LOC) with inferior economic meaning, measures dependence on programming language and their lack of counting standards. For example, programming conventions have alternating impact on LOC metric measurement. Hence, LOC measurement becomes incomparable when two different programming languages are inspected, because of the differing syntax [Fenton & Pfleeger, 1998].

According to Sommerville [2010], process quality and project size are important factors in productivity measurement. Complex software projects are more difficult to understand and implement, and require additional development effort. Variations are found between project process phases, which might be caused by differences in process stability or complexity [Raza & Faria, 2014]. Tomaszewski [2006] reports shortage of development tools to be a major productivity bottleneck in subsequent development.

3 Internal Software Quality

3.1 Introduction to Software Quality

Quality is an abstract concept and cannot be easily understood or measured [Stavrinoudis & Xenos, 2008]. There is no explicit interpretation of what high quality means [Wagner, 2013; Krasner, 1998]. In software, quality is the degree to which a set of quality characteristics fulfill the defined software requirements. When quality is an ambiguous issue and means different things to different stakeholders, it is suggested that software quality has to be specified in the context of a project [Krasner, 1998]. Quality evaluation is considered to be an important measurement for the value of software [Chappell, 2014]. There are at least six alternative definitions for software quality [ISO, 2010]:

1. The degree to which a system, component or process meets specified requirements.
2. The ability of a product, service, system, component or process to meet customer or user needs, expectations or requirements.
3. The totality of characteristics of an entity that bear on its ability to satisfy stated and implied needs.
4. Conformity to user expectations, conformity to user requirements, customer satisfaction, reliability and level of defects present.
5. The degree to which a set of inherent characteristics fulfill requirements.
6. The degree to which a system, component or process meets customer or user needs or expectations.

Garvin [1984] mentions that user satisfaction and quality are not necessarily the same thing. A software product can meet all the requirements, but it still might not satisfy the user expectations. It might be challenging to justify a good quality status just by fulfilled requirements or user satisfaction. The user's workflow or productivity could be poor, even if the used software meets the functional requirement specification. Garvin [1984] lists different approaches to quality, that are relevant at different phases in life-cycle of a product:

Transcendent

High product quality is evaluated by abstract intuition. This approach is

usually used in customer statements in requirements engineering (RE) when quality requirements are unknown.

User-based

Focus is on user satisfaction instead of technical requirements. Users can be satisfied with the affordable value of the software without being concerned about the quality.

Value-based

The value of the product is more important feature than the quality for the user. The cheaper the product, the more satisfied the user is. This approach is similar to user-based approach.

Product-based

The quantity of met requirements defines the quality. Quality can be precisely measured, but for software some metrics might not exist or are really difficult to measure.

Process

Defines quality as conformance with the specified software requirements. Sommerville [2010] notes that high-quality processes usually tend to output high-quality products. ISO 9000 standard suggests to establish quality management system in companies to ensure the appropriate quality of the products [ISO, 2005]. The standard is widely used in process approach.

3.1.1 Internal and External Quality

Internal (developer-related) quality is the quality of the source code that programmers work with on daily basis. It mainly consists of code understandability, maintainability, testability, efficiency and security [Chappell, 2014]. These quality attributes affect developers' productivity. Stavrinoudis & Xenos [2008] state that software components with low internal measurement score will also have low external measurement score for quality. Another statement is that when the internal score is low the external score is not high — only below average at maximum. High internal quality score does not guarantee high external quality. Fenton's

axiom [Fenton & Pfleeger, 1998] states that good internal software structure is expected to provide good external quality.

External (user-related) quality can be experienced through the usage of a software. For the end-users external quality means characteristics, such as usability, functionality, reliability and security. External quality measurement is generally very important in software quality management. In addition, good external measurement results contribute to the improvement of the public image of a company. The external quality discussion is kept minimal in this thesis, because technical debt are related to internal quality.

3.1.2 Diverging Quality Perspectives

Chappell [2014] defines three aspects of software quality: functional, structural and process. Functional quality describes how well a software fulfills the desired functionality. It comprises meeting the specified requirements, allowed amount of defects, performance and ease of learning and using it. Second aspect, called structural quality, measures the quality of the source code. Third aspect, process quality, significantly affects the value received by all stakeholders. Process quality has attributes such as meeting deadlines and budgets, and a reliable repeatable development process. Every quality aspect is connected to each other. When something is changed in one aspect, it affects the others. For example, tightened schedules for process improvement may increase the amount of defects in the software and decrease functional software quality.

Chappell [2014] divides software project stakeholders that care about software quality into three groups: users, development team and sponsors. Each group considers their own aspect that impacts their work most. However, each aspect has its own weight in a project and has trade-offs when emphasized. The emphasis of certain aspects vary depending on project goals and needs.

Users are interested in functional aspect of the software. However, the term "user" is a bit vague concept in software development process due to different user types [Stavrinoudis & Xenos, 2008]. Users can be, e.g., end-users, testers, and even developers. In other words, everyone who is participating in the production are users at some level. Development team cares about the structural

aspect, but also the functional quality to some degree. Well-structured source code enables developers to work and maintain the software in a desired way. Team managers require methodologies to enable manageable process and to deliver the right software for the sponsors and users. The third group, sponsors, are interested in all three aspects of quality. Sponsors are usually the business personnel that fund the development. Their interest should be comprehensive because they have to know that the project is progressing as planned. Sponsors pursue to achieve good business value, but they often ignore important technical issues and focus too much on the business related data [Chappell, 2014].

3.2 Internal Quality Measurement

Software quality is measured for various reasons in organizational perspective. Organization can create baselines and models of current practices, inspect strengths and weaknesses, or evaluate the quality of the process or the software. When a software has a specified requirements with characteristics that are relevant with the software quality, it becomes possible to measure it [Petrasch, 1999]. Without quantitive metrics, software quality measurement is subjective, because it is performed through estimator intuition or surveys. Jones [2008] states that quality has to be measurable when it occurs and predictable before it occurs. Hence, measurement framework is needed to describe the baselines that are then analyzed to form an understanding of dependencies between processes and events. Analysis of this knowledge generates a model that needs evaluation to enable the creation of model for prediction and instruction.

Software measurements give usable information about the quality of development and assist in defect prevention. The international quality standards define quality measurements to enable organizations to achieve appropriate product quality [ISO, 1997; ISO, 2000]. However, a general problem in measurement is the lack of guidelines that help to accomplish the right process. The end-product quality of any software project is heavily connected to measurements that are used to monitor software quality factors [Stavrinoudis & Xenos, 2008]. Measurement is important for process (e.g., planning or development), products (e.g., deliverables or documents) and resources (e.g., personnel or materials) [Scotto *et al.*, 2004].

Quality measurement is often subjective, hence problematic when not executed through practices that are well managed [Moses, 2009]. There are also two quality factor groups concerning all entities: directly and indirectly measurable factors. Internal quality is measured directly and has objective results. Internal measurements are performed automatically to source code with internal software metrics. Internal quality measurement is fast, easy and automated process and it could even be performed on unfinished software projects. Error frequency is considered to be minimal and the measurement results are objective. However, the results are low-level information and difficult to be interpreted and utilized in practice.

External quality is measured indirectly and is subjective. External measurements usually require the software to be finished and evaluated by the users before measurement [Stavrinoudis & Xenos, 2008]. Stavrinoudis & Xenos [2008] list that heuristic evaluation performed by experts [Nielsen, 1994], performance measurements [Dumas & Redish, 1999; Rubin, 1994] and the user-perceived software quality measurements [Xenos & Christodoulakis, 1997] are the most common external quality measurement techniques. The downside of external quality measurement is its high costs [Stavrinoudis & Xenos, 2008].

Quality measurement goals can also be viewed from different perspectives, such as customer, manager or developer [Sun, 2011]. Bohnet & Döllner [2011] emphasize the importance of monitoring, management and mitigation of internal quality during development and maintenance. Maintainable software is better achieved through these activities due to high non-realizability of internal quality, especially amongst management. To make the internal measurements worthwhile for any project, it is important to acknowledge that the costs of internal measurements are significantly lower compared to external measurements [Stavrinoudis & Xenos, 2008].

3.3 Internal Quality Metrics

Software quality is often measured both qualitatively and quantitatively. Quality metrics are used when quantified and measurable data is extracted from software. The purpose of software metrics is to assist software developers to inspect source code and enable them to improve their practices. To get meaningful results from

measurement, the targeted source code fragments must be large enough. Specific software development languages and frameworks require an appropriate set of metrics to be used for measurements [Stavrinoudis & Xenos, 2008]. Software quality metrics have been proven to reflect internal software quality and are widely used in software quality evaluation methods to identify those parts of software that require re-engineering [Boehm *et al.*, 1976]. Common quality metrics include the following [Singh, 2013]:

Lines of Code (LOC)

Measures the size of software. Smaller size means better understandability.

Cyclomatic Complexity (CC)

Measures the complexity of class methods. Lower complexity score means higher quality.

Coupling Between Objects (CBO)

Measures the level of coupling associated with the class that is being measured. Less coupling means less complexity.

Lack of Cohesion of Methods (LCOM)

Measures the level of encapsulation in a class. Higher cohesion score means less complexity and higher reusability.

Weighted Methods per Class (WMC)

Measures the sum of complexities in a class that are individually measured with CC. Lower weight means better maintainability, reusability and less complexity.

Response For a Class (RFC)

Measures the level of communication that a class has with other classes. Fewer responses mean less complexity and effort in testing and better understandability.

Maintainability Index (MI)

Measures the ease of software maintenance through other metrics. Higher index means higher maintainability and quality.

Number of Children (NOC)

Measures the number of children that a class has. More children means better reusability, but higher complexity and testing effort.

Depth of Inheritance Tree (DIT)

Measures the depth of class inheritance hierarchy. Greater depth means higher reusability, but also higher complexity.

A definition that has been commonly used to define quality is the density of post release defects in software, which is indicated by the number of defects per thousand LOC (KLOC) [Diaz & Sligo, 1997; Fox & Frakes, 1997]. Krishnan & Kellner [1999] and Gaffney [1984] report that LOC is the most suitable estimator for the amount of errors in code. According to Banker & Slaughter [2000], data complexity and software volatility are significant predictors for errors. Data complexity is the number of data elements per unit of application functionality. Software volatility is the frequency of enhancements per unit of functionality in certain time frame.

3.4 Quality Models for Evaluation

Metaphors are often used in software design process to aid understanding [McConnell, 2004]. Using metaphors in this way is called "modeling". In order to understand and measure quality, a number of quality models and standards are specified to help software development organizations to build software with appropriate level of quality. Quality models divide the concept of "quality" into different quality factors (aka characteristics or attributes). These models are mainly used during requirements analysis as checklists [Wagner, 2013]. Descriptions for the different quality characteristics are listed in Appendix A. There are multiple quality models with various characteristics and most of the characteristics will overlap although they might be named bit differently.

McCall's Model

McCall's quality model [McCall *et al.*, 1977] focuses on certain software

quality factors concerning developers' priorities and users' views. The software product is viewed from three different perspectives. Revision perspective measures the ease of modifiability. Transition perspective measures the ability to adapt to other environments. Operations perspective measures the characteristics of operation. McCall's quality model structure is shown in Figure 3.1.

Boehm's Model

Boehm's model [Boehm *et al.*, 1978] is similar to McCall's model by having a hierarchical categorization around high-level characteristics. It concentrates on defining software quality qualitatively by a given set of attributes and metrics. The structure is described in Figure 3.2.

FURPS Model

Grady [1992] introduced a model called FURPS, which was later extended by IBM to FURPS+. FURPS stands for functionality, usability, reliability, performance and supportability. FURPS is also similar to the previously introduced models but it has not been that popular or used.

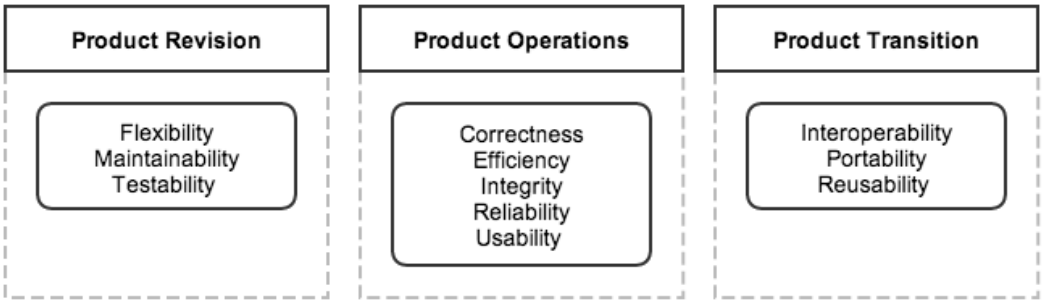
Dromey's Model

Dromey's model [Dromey, 1995; 1996] focuses on the relationship between characteristics and sub-characteristics of quality. It attempts to connect software properties with software quality attributes [Singh & Kannojia, 2013]. The quality model is designed to be dynamic and product based, that transforms to fit the product context. The model is shown in Figure 3.3.

ISO 9126

ISO 9126 is a derivation of McCall's quality model that was introduced in 1992 [Singh & Kannojia, 2013]. ISO 9126 [ISO, 2001] standard is the most well-known taxonomy for software quality and recognizes the existence of internal and external quality characteristics. The standard describes quality as "the totality of features and characteristics of a product or service that bears on its ability to satisfy given needs" [ISO, 2001]. The hierarchy is presented in Figure 3.4.

Figure 3.1 The structure of McCall's quality model



ISO 25010 (SQuaRE)

ISO 25010 [ISO, 2011] was formed in 2011 and it replaces the ISO 9126 standard. It has eight quality characteristics with sub-characteristics as described in Figure 3.5. The quality specification is called "SQuaRE", which stands for "Systems and Software Quality Requirements and Evaluation". SQuaRE series of standards is the most applicable and long-running standard to software quality control.

Figure 3.2 The structure of Boehm's quality model

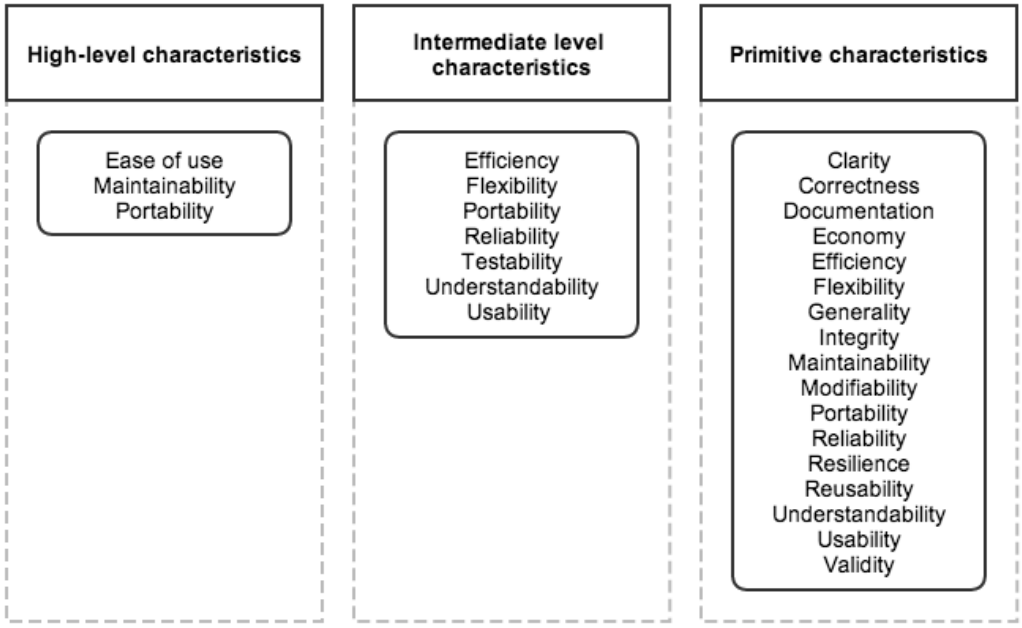


Figure 3.3 The structure of Dromey's quality model

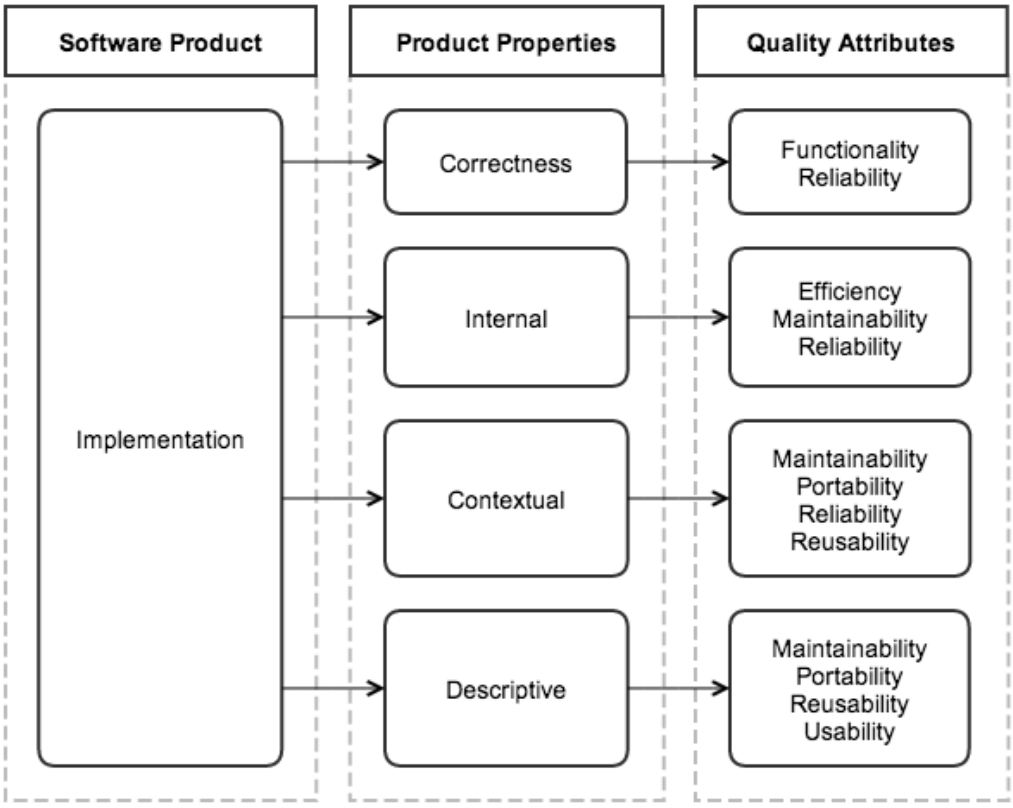


Figure 3.4 The structure of ISO 9126 quality model

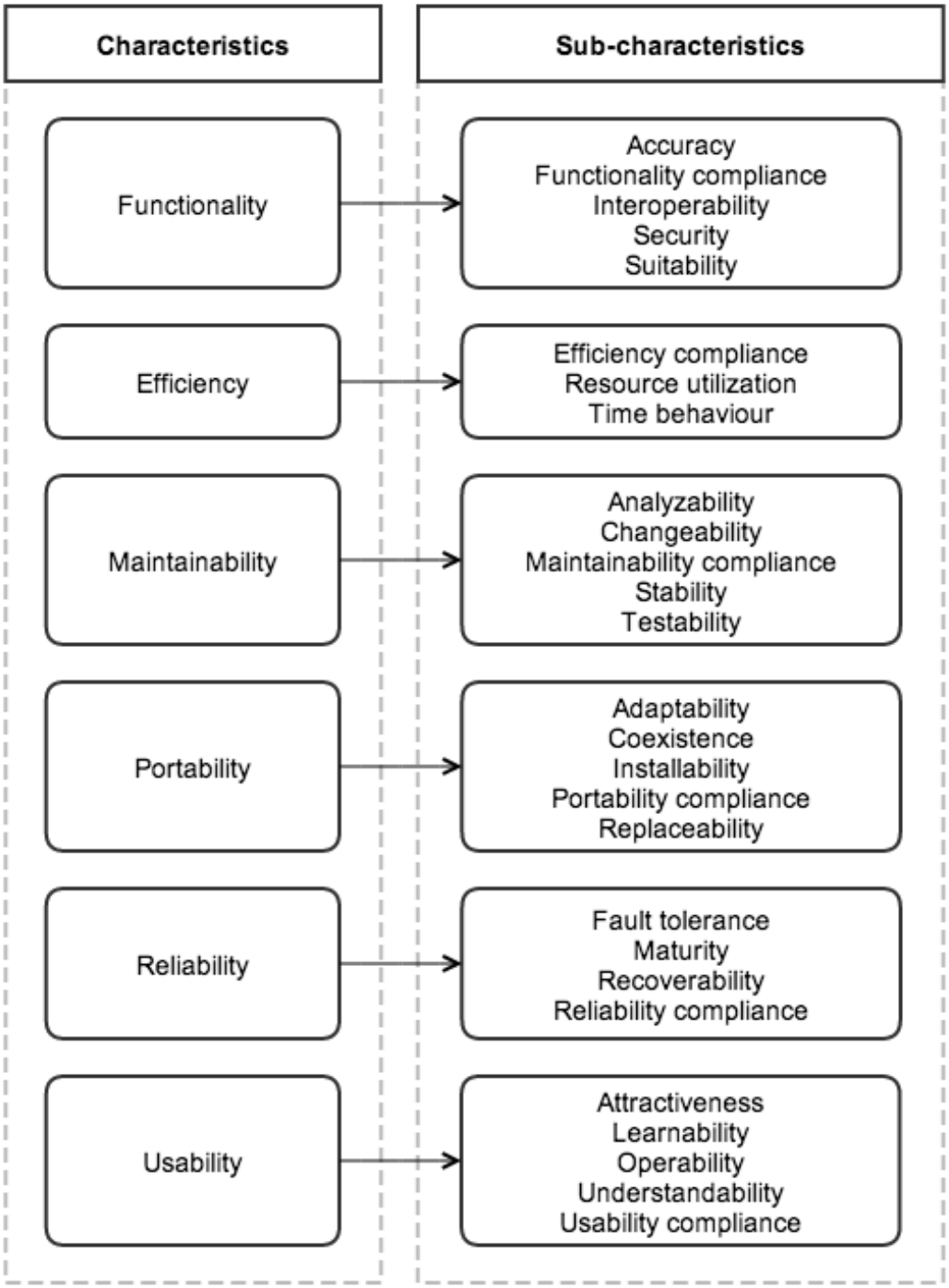
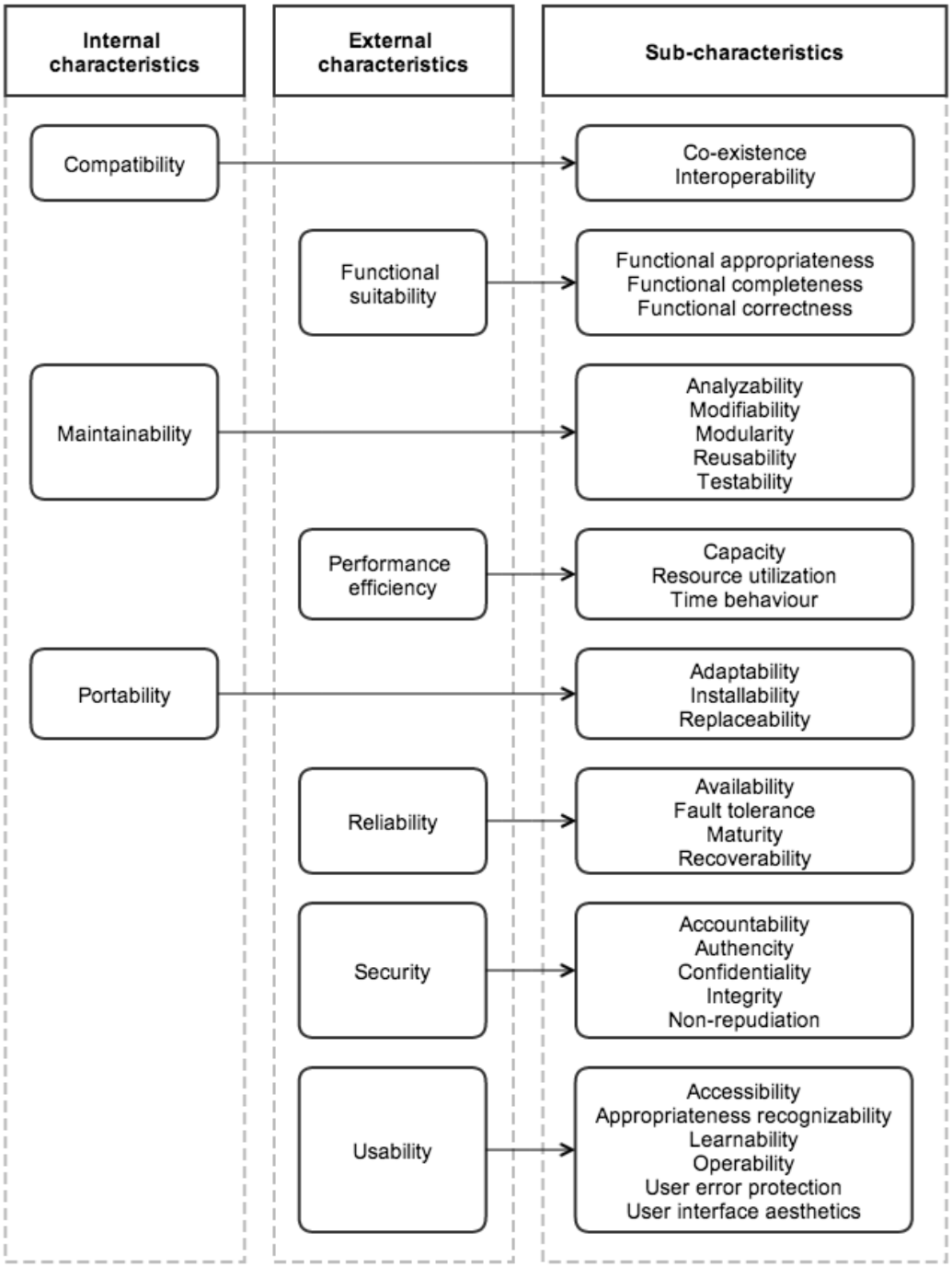


Figure 3.5 The structure of ISO 25010 quality model



4 Technical Debt

4.1 Introduction to Technical Debt

Cunningham [1992] introduced a metaphor "technical debt" to describe design choices and the necessity of refactoring, because design choices have direct impact on software quality. Technical debt specification has been broadened afterwards by several authors, such as Fowler [2003], McConnell [2008] and Sterling [2013]. The metaphor has been extended to include architecture and design by Kerievsky [2004]. Curtis *et al.* [2012] define technical debt as the cost of fixing structural quality problems in production code, that organization knows must be eliminated to control development costs or to avoid operational problems. Ergin [2013] presents the following type categorization for technical debt:

Strategic Debt

Organization pursues to gain an advantage in the market by releasing the software product before its competitors.

Tactical Debt

Organization decides to accumulate technical debt and remediate it later, for example, when an organization aims to meet planned deadlines.

Inadvertent Debt

Organization makes cost savings by hiring unexperienced newcomers to develop the project. However, newcomers' lack of development know-how incurs technical debt to the project.

Unavoidable Debt

Organization is demanded to make changes to developed software, for example, by the government.

Incremental Debt

Development team accumulates technical debt in development through poor development practices (e.g., poor experience, knowledge or motivation).

Design Debt

Development team takes shortcuts in design or implementation, or design too much upfront. Solutions usually end up being suboptimal.

Code Quality Debt

Software quality is poor because of the amount of software defects or crashes, and complex and poorly structured source code.

Testing Debt

Development process lacks automated testing and too much testing is performed after software changes.

Knowledge and Documentation Debt

Development team is lacking in system knowledge and there is a high risk of extensive debt, for example, if key system personnel leave the organization.

Environmental Debt

Development process, hardware and infrastructure has issues and too many operational tasks are performed manually.

Different types of debt occur in different phases of development [Rothman, 2006]. Some types are intentional, some unintentional and some are resulting from another debt. Different debt types might need different measures and approaches for management, because of their varying attributes. In this thesis, the term "technical debt" is considered to include various types of debt, so the technical debt is discussed in broader extent.

Technical debt is accumulated when the internal quality of a software is compromised through intentional or unintentional actions. *Intentional debt* is incurred through conscious decisions and actions for business reasons. One of the reasons for intentional technical debt is that it can enable strategic advantages for organizational goals [Klinger *et al.*, 2011]. Intentional debt consists of requested changes, refactoring and debugging that are deferred to future release [Snipes *et al.*, 2012].

Unintentional debt is commonly caused by inexperienced or unskilled individuals in a development team. Debt also incurs when appropriate design standards and

practices are neglected. Developers could use poor design practices and methods on daily basis, so that maintainability decreases over time and affects productivity of the whole development team. Concerning the impact on internal quality, unintentional debt could be avoided through informative development team feedback on the decisions related to prioritization strategies [Theodoropoulos *et al.*, 2011]. Unintentional debt is typically more problematic type of debt than intentional debt and is considered to be a challenge in software industry. Different sources, motivation and management of technical debt is found to be more complex than simple tradeoffs made by technical architects [Klinger *et al.*, 2011].

Every software project involves certain level and form of technical debt, which always has some negative effects [Higgs, 2011]. Gartner[1] reports that global IT debt was $500 milliard in 2010 and potentially rises to $1 billion by 2015. However, Buschmann [2011] states that it is often beneficial to incur technical debt and pay it off later and reminds that paying off technical debt does not always provide the best value. In some cases, it could be beneficial to let the debt retire with the software, if the software is planned to be a short-term solution.

4.2 Sources of Debt

Klinger *et al.* [2011] state that rushed production to meet deadlines and decisions made in other projects that affect the project in question cause debt. In addition, other debt sources are such as acquisition, new alignment requirements or changes in the market ecosystem. According to Klinger *et al.* [2011], management practices and decisions are often informal, which incurs technical debt. Technical decisions are made without appropriate analysis concerning their impacts and risks.

When technical architects make bad design decisions, they cause extensive development issues in the future when the architectural foundation lacks flexibility, functionality or cohesion. Architects are responsible for the implemented technical features in code-level, but there are numerous stakeholders to ensure project success concerning testing, brand strategies, legal, marketing and so on [Williams *et al.*, 2010].

[1] `http://www.gartner.com/newsroom/id/1439513`

4.2.1 Poor Practices

Occasionally, managers and developers need to make decisions to gain short-term benefits at the price of decreasing internal quality and long-term concerns such as maintainability. Source code that has been written by taking shortcuts might be fast to work with at first, but the shortcuts will slow down the development process in long-term. It is also risky to later make changes to poorly structured source code, because one small change can break several other software components.

However, the usual reason behind bad software design is the incompetence of an individual development team member that unintentionally accumulates technical debt. When the software development is not controlled properly, unintentional debt can creep in. Unintentional debt, such as spaghetti code, loose user input validation practices or insufficient unit testing could cause tedious issues. For example, if faulty input data is allowed and saved into the system, the data quality decreases and can cause critical issues to the business that is using the software. Hence, neglecting development process control causes decreased system accuracy and degrades security [Theodoropoulos *et al.*, 2011].

Lack of refactoring is a common issue that degrades source code quality and lets technical debt to accumulate. The benefits or costs of refactoring are not usually quantified, which makes the justification of it difficult when communicating with management [Zazworka *et al.*, 2011]. Refactoring effort and impact estimation is difficult in general, so refactoring might be unattractive and easily avoided.

4.2.2 Lack of Collaboration

Software development is social activity that requires collaborative planning, brainstorming and designing. Solving problems can be difficult, and wrong decisions and designs are difficult to avoid. Through communication and collaboration, ideas, designs and mistakes are transformed to solutions, that pushes the project to go forward.

The decisions that incur technical debt are often made by stakeholders that are unaware of the consequences of their decisions [Klinger *et al.*, 2011]. The decisions

always affect some other part of stakeholders. It is common that business, operational and technical stakeholders have difficulties in communication, that further cause issues in projects [Williams *et al.*, 2010]. For non-technical stakeholders, there might be no suitable way to communicate with the technical department about technical issues. Hence, organizations often lack the global view and communication channels to be able to optimize complex technical debt issues.

According to Theodoropoulos *et al.* [2011], technology departments are said to be out-of-sync with the rest of the business in organizations, because of the historical differences in perspectives. By enabling stakeholders to evaluate and manage quality issues consistently across the enterprise will also enable more effective collaboration for meeting common goals.

4.2.3 Technical Ignorance

Business executives, technology leaders, risk managers and end-users have interest in project's well-being, but are not usually interested in technical issues or internal quality [Theodoropoulos *et al.*, 2011]. Business partners should always understand the quality implications of their business decisions. Klinger *et al.* [2011] report that decisions are rarely quantified appropriately to be able to be monetized. Technical debt quantification is difficult, because it is relative to a defined set of goals, requirements and stakeholders in a project.

4.2.4 Overoptimistic Estimation

Technical debt is often a symptom of objectives that are generated through overoptimistic estimation. Ktata & Lévesque [2010] report that estimations done in agile environments are also set as objectives for development (Parkinson Law [Ottaviani & McConnell, 2006]). These estimations are performed with certain error rate percentage but can be far more off than the percentages imply. To emphasize this, Austin [2001] mentions that schedule pressures cause developers to compromise quality. Over time the technical debt is incurred over process iterations at high rates and the development becomes very stressful for the developers.

4.2.5 Outdated Technologies

Various technologies and development components keep evolving the same way as the developed software itself and this means continuous need for adaptation. Outdated technologies or development components can raise critical development issues over time. From the development perspective, issues related to outdated technologies and components can overcomplicate or even block the development process. Outdated technologies or components cause issues like component incompatibility or lack of required features. Compatibility issues require additional effort and cause additional costs. Technical debt accumulates when outdated technology causes issues that are solved by taking shortcuts in development to "save on budget".

4.2.6 Aggressive Product Growth

When a software organization is facing an aggressive growth scenario, the rapidly increasing business demand forces the organization to develop new features. When the growth of the organization is increased aggressively, the technical debt accumulates in increased pace. Technical debt rapidly decreases maintainability and development robustness, furthermore making new implementations impossible [Ramasubbu & Kemerer, 2013]. The opposite to aggressive growth is delayed growth, in which the software has low utilization level.

4.3 Consequences of Debt

Technical debt affects several quality characteristics of software, that are listed in Appendix A. Even a short-term negligence on technical debt management can lead to a point where the development team is mainly dealing with software defects and no new features are implemented [Snipes *et al.*, 2012]. Development team members might not be aware of the state of accumulated debt. Furthermore, this means that they do not know that they cannot actually manage any feature requests. This scenario is called "Death Spiral". Death Spiral eventually makes the development of the software unsustainable. This scheme can cause team members to quit their job and/or incur major financial issues to organization's

business. Technical debt has severe impact on software quality through negative effect on team's productivity, collaboration and overall transparency [Ktata & Lévesque, 2010].

When an excessive amount of technical debt accumulates, software development team will eventually request that the software must be redesigned. Usual response to this is to deny the need in order to save resources. In a way or another, the management must and will agree to redesign to regain appropriate developer productivity [Martin, 2009]. As a reaction to decreased productivity in a project, management usually recruits more developers to the project to fix a deficit [Martin, 2009]. Unfortunately, the effect is the opposite, because the newcomers are not familiar with the system design. Under heavy pressure, the productivity of the development team decreases even further.

When software features are deferred, the operability characteristic is affected. When the users are suffering from decreasing operability, the accuracy of required functionality is decreasing [Theodoropoulos *et al.*, 2011]. As a result, users are encouraged or even forced to create workarounds when required features are inaccurate, unusable or missing. This means that incoherent working habits are created and the security of the data in the system is compromised.

Poor user input validation risks a whole software system by enabling user to input erroneous data. Bad data introduces quality defects on processes or systems that interact with it. The reason for this is the inaccurate or incomplete data that is not reliable. When a system is generating or utilizing bad data, system integration becomes very hazardous and could cause wider scale issues to other systems. Bad data generally decreases software's functional accuracy and interoperability [Theodoropoulos *et al.*, 2011]. From developer's perspective maintainability is affected through decreased changeability and testability.

Technical debt has principal and interest costs. Principal costs occur when a defect is fixed on detection. Interest costs keep accumulating until the defect is fixed, which is the case when the fixing of a defect is postponed. Interest includes costs that are caused by additional effort related to technical debt [Snipes *et al.*, 2012]. In addition to principal and interest costs, technical debt causes risks to business when critical defects cause unexpected issues in software operation. For example, if data gets corrupted or fatal system error halts the system execution,

the required maintenance procedures will incur additional costs.

To define principal costs for technical debt reduction the structural quality has to be analyzed. Defect severity analysis and high-severity defect fixing prioritization are needed to estimate costs [Curtis *et al.*, 2012]. The principal cost is estimated through the number of critical defects, time required to fix each defect and the cost for fixing a single issue. The time needed to fix an issue consists of analysis, comprehension, correction planning, side-effect evaluation, fix implementation and release operations.

Curtis *et al.* [2012] present an example of principal calculation. The equation

$$\text{Principal} = \sum_{i=1}^{n} (v_i \cdot f_i \cdot t_i \cdot c_i) \tag{4.1}$$

is in generalized form, where n is the amount of severity categories, i represents a violation severity category, v_i is the number of violations, f_i is the percentage of violations that must be fixed, t_i is the time that fixing a single violation takes in hours, and the variable c_i is the average cost of fixing per hour. Estimates resulting from Equation 4.1 should be treated as lower bounds [Curtis *et al.*, 2012]. Estimations also vary broadly and depend on used programming language, for example. Curtis *et al.* [2012] calculated an average cost of 3.61 dollars per line of source code for technical debt existence in their example.

5 Refactoring

5.1 Introduction to Refactoring

The word "refactoring" is formed from word a "factoring" which was used in structured programming [McConnell, 2004]. Factoring means decomposition of software to manageable components. Cunningham and Beck were the first ones to recognize the importance of refactoring [Fowler *et al.*, 1999].

Refactoring is a process that aims to improve the quality of the source code of a software system without altering the behavior experienced by the users. The goal is to restructure software source code to be more readable, understandable and cheaper to maintain. Refactoring process mainly targets to reduce the amount of source code, but it can also increase it [Fowler *et al.*, 1999]. Also, refactoring is assumed to improve non-functional quality aspects, e.g., extensibility, modularity, reusability, complexity, maintainability and efficiency [Mens & Tourwé, 2004].

Refactoring helps to improve developer productivity and reduces produced software defects [Mens & Tourwé, 2004; Fowler *et al.*, 1999]. Refactoring is the key element in the whole process of software development [Fowler *et al.*, 1999]. Improved internal code quality enables developers to be productive with their daily work. Refactoring can also be used in source code familiarization when new developers join a project. Getting familiar with the source code usually takes months, so refactoring is a good way to learn. The role of refactoring becomes more significant if the source code quality or the documentation is poor.

Refactoring is an alternative to upfront design and is used in modern agile software development practices. Fox example, in XP methodology, the working software is written quickly and refactored afterwards to meet the quality standards. Upfront designs will always be changed multiple times throughout a project to meet the specified requirements [Fowler *et al.*, 1999]. So, the design does not have to be right the first time. Refactoring can and should be done continuously during the project to different parts of software, such as source code design and structure, database structure or documentation. Refactoring decreases the possibility to make mistakes in the code by cleaning up the cluttered and unreadable code to improve maintainability and therefore enable the developers to do their

work properly.

5.2 Code Smells

A common problem for software engineers is to discover when and where to apply source code refactorings [Stroggylos & Spinellis, 2007]. Fowler *et al.* [1999] state that this problem is usually handled with human intuition and method called "Bad Smells" or "Code Smells". Code smell is a hint that indicates that source code might be poorly designed and needs to be refactored for appropriate level of maintainability. Moreover, there are software metrics that can identify areas that benefit from refactoring and various tools that support the human intuition and help to refactor [Stroggylos & Spinellis, 2007]. Code smell indicates a possible refactoring opportunity within or between classes. The following code smells indicate refactoring opportunities *within a class*:

Duplicated Code
: Code structure has one or more duplicate in the code.

Long Method
: A class method is too long.

Large Class
: A class has too much responsibilities and can appear as too many instance variables.

Long Parameter List
: A class method has too many parameters.

Switch Statements
: Switch statements lead to duplication. Polymorphism should be considered instead.

Speculative Generality
: Obsolete generalization when it is not needed.

Temporary Field

A field of an instance that is set only in certain circumstances. Fields should be always used.

Comments

Source code should explain clearly what it does, hence code comments are meant to explain "why" instead of "what".

The following code smells indicate refactoring opportunities *between classes*:

Divergent Change

When a single change requires multiple other changes in a class instead of targeting one method.

Shotgun Surgery

When making a change somewhere requires a lot of minor changes in many other classes.

Data Clumps

Some data items existing together in lots of places.

Parallel Inheritance Hierarchies

A case of shotgun surgery, in which a new parallel subclass is required to be added due to new subclass addition.

Feature Envy

A method that processes more other classes' features that the one's it is located in.

Primitive Obsession

Primitive types are overused in software instead of using small class objects (e.g., tel. number, zip code or currency).

Lazy Class

A class that should be eliminated, because it is not doing that much in the software.

Message Chains

When a method asks for an object for another object, and again for another object forming a long chain.

Middle Man

A class is delegating many methods of another class.

Alternative Classes with Different Interfaces

Similar functionality between classes.

Inappropriate Intimacy

Classes manipulate others' features too much.

Data Class

Classes with fields, getters and setters only. Other methods are manipulating these extensively.

Incomplete Library Class

Too much trust is put on third-party libraries and it may cause troubles when extending functionality, because the libraries are really technically unknown for the developer.

Refused Bequest

Subclasses inherit methods and data from parent classes, but rarely use them.

Code Smells are broadly used to aid the refactoring process [Mens & Tourwé, 2004]. However, very few studies report on the effect of using code smells and it is not known whether the code smells are effective way to guide refactoring and improve source code [Zhang *et al.*, 2011]. Zhang *et al.* [2011] report that most studies focus either on one or two code smells or all 22 of them. They also report that "Duplicated Code" smell was the most studied. However, Zazworka *et al.* [2011] present that "God Class" ("Large Class") is the most commonly appearing code smell and refactoring it requires most modifications compared to other smells. Monden *et al.* [2002] found that in some situations "Duplicated Code" improves reliability. Other authors suggest that every duplication does not

necessary need refactoring because of this. Some situations require that refactoring costs and risks are measured and compared against the expected gains in maintainability. Zhang *et al.* [2011] also list that some studies indicate that "Large Class", "Large Method" and "Shotgun Surgery" code smells are significantly associated with software faults of all severity levels. Code smells that were not associated with any faults were "Data Class", "Refused Bequest" and "Feature Envy".

5.3 Performing Refactoring

Refactoring is applied with short, continuous and controlled bursts during development to restructure the source code. Refactoring is a part of every day software development to ensure cohesive production of good quality code [Fowler *et al.*, 1999]. Good opportunities to refactor are when performing modifications or maintenance, adding new classes, methods or routines, and fixing bugs. Compiler, or similar, logs are also a good way to get indications of refactoring opportunities. After a refactoring opportunity is detected, its scale has to be analyzed. The scale of refactoring can differ widely. For example, renaming a variable is a small task, restructuring a method is a medium task and architecture redesign is a large task. The larger the modification scale is, the more risks it involves.

When deciding what to refactor, it is important to target complex components that usually cause the most problems in the software system. However, this kind of refactoring is usually avoided because developers tend to hate or fear complex source code that might end up malfunctioning after the procedure [Fowler *et al.*, 1999]. Concerning changes that improve code quality, the LeBlanc's Law states: "later equals never" [Martin, 2009]. In other words, refactoring is meant to be performed continuously. There is also a "refactoring rule of three" defined by Roberts [Fowler *et al.*, 1999]:

> "The first time you do something, you just do it. The second time you do something similar, you wince at the duplication, but you do the duplicate thing anyway. The third time you do something similar, you refactor."

The quality of software design is improved only when refactoring is used appropriately. In addition to improved internal quality, refactoring decreases the amount of introduced bugs and makes testing easier [Fowler *et al.*, 1999]. There are tools to automate refactoring tasks, but they are quite risky to use and can cause unexpected results. Manual refactoring done by the developer usually brings better results when the source code is inspected and understood.

Although refactoring increases the software response times, it also makes the software performance optimization easier through improved source code readability and structure. When source code is clean and well-structured the optimization options are easier to see. However, the optimization process should be done at the end of the development process in its own phase [Fowler *et al.*, 1999]. Performance optimization is quite similar process to refactoring, but it focuses on improving the software execution and response times. Developers pay great attention to keep the software performance high while implementing components, but the lack of effectiveness is caused by messy source code.

5.4 Risks of Refactoring

Programmers often use or claim to use refactoring to improve the internal quality, but the quality metrics usually indicate that the actions have had the opposite results [Stroggylos & Spinellis, 2007]. Developers also often state that software quality is a top priority, but in reality they are driven by schedules. It is common that project managers depend on schedules and pay attention to development velocity without paying enough attention to the internal quality issues. As Fowler *et al.* [1999] state, programmer does not need a permission to refactor. Even though developers can implement solutions their way, there has to be a cohesive vision and understanding within the development team. However, refactoring should not be performed when project schedule is tight and a deadline is approaching [Fowler *et al.*, 1999]. If refactoring is not applied properly it can decrease understandability instead of improving it [McConnell, 2004]. There is also a high risk to introduce bugs when source code is modified under pressure.

Whenever refactoring is performed, it is essential to acknowledge that an ongoing refactoring task must be completed before moving on to the next one. The

refactored code must also be accessible after refactoring is completed, so it does not introduce any new issues. Instead, new refactoring opportunities that are discovered during certain refactoring process should be noted down for future processing. Any refactoring that is left unfinished is a risk that can introduce more issues. The risk level of a refactoring task affects how the refactoring is approached and executed [McConnell, 2004]. To reduce risks, constant testing and reviewing helps to control the refactoring process and avoid mistakes. Formal guidelines and developer know-how are also important factors that help to reach good refactoring results.

Kernighan & Plauger [1978] state that bad quality source code should never be patched or documented — it should be rewritten. When a system-wide refactoring is planned, it is essential that the effects of individual refactoring operations are monitored [Fowler *et al.*, 1999]. There is a huge risk that some partial refactorings cause a defect that cripples the system operation. Checklists can be used to help to keep track of the refactoring steps. If refactoring modifications does not work as they are supposed to, there must be the ability to revert the changes.

Refactoring source code to more readable form usually incurs software performance loss as a tradeoff [Fowler *et al.*, 1999]. Although, performance issues are often located in small segments of the code. There are just a few segments that cause the majority of performance loss in software. It is also good to acknowledge that most part of the source code is rarely executed, implying that the time spent optimizing these fractions of the code is expensive. Ron Jeffries, an XP development methodology professional, stated that performance should always be measured instead of guessed or estimated [Fowler *et al.*, 1999].

Databases could also be affected by refactoring. Business software systems are often coupled tightly with their databases, which makes database architecture changes challenging. Changes to database structures require data migration, that is time-consuming task to accomplish. However, layers can be implemented between databases and the actual object models to achieve flexibility. Higher flexibility always means higher complexity [Fowler *et al.*, 1999], and thus, more written code.

Refactoring should not be done in a software system if there is no access to published application programming interfaces (API) [Fowler *et al.*, 1999]. For

example, when a method is renamed, it means that every other software that uses the method has to be modified to use the new method name when calling the method. The solution to enable the refactoring in this kind of situation is to leave the legacy functionality available and mark them as deprecated. The refactored functionality is then implemented in parallel. This way the interface users have time to react to these changes and adapt to use the new refactored version of the function.

6 Enabling Sustainable Productivity

6.1 Motivation for Internal Quality Investment

This section summarizes some points to answer our first research question: *"Is high internal quality essential in software projects? If so, why?"*. It is often said that quality cannot be justified to be essential in every project, hence this topic is quite difficult to answer, but there are some fundamental points to any software project to meet that quality promotes.

6.1.1 Project Success

The simplest and most foundational requirement for any project is success. To be able to measure when a project meets its requirements, there has to be specified and agreed scope for it. Project scope is an agreement and understanding between the customer and the development. Then the time and cost constraints are estimated for the project from the specified scope. Project success requires that the defined scope is comprehensive, because inadequate scope definition means inaccurate estimations, and furthermore incorrect costs and schedule. Therefore, comprehensive project scope is a foundational requirement for project success.

The quality of a project is the quality of the used process and the people involved [Fenton *et al.*, 2004; Fenton *et al.*, 2008]. Generally, project success is all about the correctness and quality of a software product [Sarigiannidis & Chatzoglou, 2013]. According to Krishnan *et al.* [2000], people quality alone strongly estimates the resulting software quality. Hence, establishing a good quality process for development and training for personnel are fundamental drivers for success. High process and people quality also decrease risks and promote success [Ould, 1999]. However, project success does not depend solely on successful development, because it is heavily affected by, e.g., brand strategies, legal and marketing [Williams *et al.*, 2010].

Various quality factors are significant for success in software development and require great attention in any project. This means that the internal quality of software is an important issue when it affects the development productivity sig-

nificantly. Relevant quality factors are often neglected in culture and literature of software requirements [Gilb, 2000] and internal quality issues often leave without required attention [Stroggylos & Spinellis, 2007]. Business stakeholders concentrate more on the irrelevant quality aspects in general [Theodoropoulos *et al.*, 2011]. However, high internal quality does not guarantee high external quality [Stavrinoudis & Xenos, 2008]. Hence, internal quality improvement cannot be unambiguously proven to generate good external quality.

Development success depends heavily on architectural design, since the complexity and size of software is increasing [Chen *et al.*, 2010]. Flexible architecture makes maintenance easier, but increases complexity and code base, which makes maintenance more expensive [Fowler *et al.*, 1999]. Harter & Slaughter [2000] state that source code complexity decreases software quality, but Gaffney [1984] reports that complexity does not affect defects significantly. Complexity affects development because it makes source code much harder to understand and often harder to read because of complicated logic and structure (more written code). As software reuse decreases the code base [Boehm, 1988], it becomes an interesting topic for future software development and research.

6.1.2 Staying on Budget and Schedule

Any software project requires accurate estimations to stay on budget and schedule. Software projects being very dynamic with all the changing requirements, estimations are difficult to perform. According to Popli & Chauhan [2014], there are currently no common mathematical formula to accurately calculate these estimations. Jørgensen [2004] reports expert estimation to be a dominant strategy in software project effort estimation. There are no substantial evidence supporting the superiority of estimations that are performed using models over the expert estimates. When performing informal estimations, it might be familiar to end up estimating what the management wants to hear, instead of having the courage to provide more realistic estimates. The accuracy of estimation is critical, because given estimates are often set as objectives for the project [Ktata & Lévesque, 2010].

For cost estimation, the inaccuracy of initial estimation can be up to 4 times the calculated estimation [Boehm, 1981]. The cost of software can increase due to

various reasons, such as poor project planning, changing requirements, development issues and changes or assumptions that are too optimistic. Ramasubbu & Balan [2012] summarize some root problems for cost estimation, including missing required information and lack of experience with the used tools. Missing information introduces issues when there are no data to input to required calculation variables in initial phases of a project. Estimations made with lacking estimation tool experience usually result in low quality estimates. Therefore, accurate estimations are a critical part of business because they enable to adjust the project goals. If costs or required effort are projected to be higher than planned, the project features or budget can be adjusted accordingly.

Costs related to software defect fixing are a major part of software costing. According to Glass [2001], around 40% of software project costs are caused by software defect detection and removal. The earlier a defect is detected and fixed, the fewer it costs to the business [McConnell, 2004]. Good quality software design decreases additional costs, because it prevents re-engineering [Stroggylos & Spinellis, 2007]. Re-engineering also decreases productivity, so it should be avoided by utilizing risk management, prototyping, incremental development and modern programming practices [Boehm, 1988]. However, quality control is an expensive investment itself, but it is a key element to optimize productivity, schedules and customer satisfaction [Sun, 2011]. When software costs are understood and controlled, it inevitably requires an organization to understand and control various aspects of software quality [Boehm, 1988].

Costs of defect fixing are divided into six categories, that are investigation, modification, workaround, customer support, patching and validation [Snipes *et al.*, 2012]. Snipes *et al.* [2012] report that investigation cost estimate is between 50–70% of the cost of fixing a defect. Modification costs are estimated to be less, from 10% to 15% of the total defect fixing cost. Validation costs are estimated to be 20–30%, but they usually spread across all cost categories. It is shown that defect investigation is approximately over half of the fixing costs. This means that good internal quality decreases project costs. Software defects has to be detected and fixed as early as possible in development process, because it prevents fixing cost escalation [McConnell, 2004]. Snipes *et al.* [2012] report that there are no findings that deferring a defect would increase or decrease overall costs, but patch requested by a customer on a deferred defect incurs even higher costs.

The problem with small organizations is the lack of realistic knowledge on costs, because there are no established budget or monitoring for software quality [Porter & Rayner, 1992; Plunkett & Dale, 1983]. According to Krasner [1998], cost of quality models are rarely used, which could mean the preference of informality. However, according to Jørgensen [2004], there is no proof that cost models are better than subjective estimation. The problem with the usage of economic models is the lack of common data in software industry [Wagner, 2013]. Hence, it is difficult generate estimates when there is no data for estimations to be based on. As quality control is an expensive investment [Sun, 2011], it is possibly the most common single reason for informal quality control in small organizations.

6.2 Foundations for Productive Development

This section presents certain foundational elements concerning the process and people quality to answer the research question: *"How to promote productive development?"*. Productivity is affected by many factors from individuals to the process, allowing many ways to improve productivity.

Process quality is an important entity to enable productive working environment, and the need for software process improvement has been identified [Chrissis *et al.*, 2003]. Process improvement increases the product quality [Banker & Slaughter, 2000; Sun, 2011] and business value [Chrissis *et al.*, 2003; McFeeley, 1996]. Process improvement might sometimes mean changing the whole process, instead of modifying the existing one. For example, the problem might be that the development methodologies that are used in a project, are not dynamic enough to be able to manage the continuous changes. Processes can be measured for size, effort, schedule and cost under successful performance. The measurements include data concerning the time, size and defects in a process. The process of any project should always be specified to meet organization's business goals and intents [Singh & Kannojia, 2013].

Raza & Faria [2014] note that individuals and teams can be helped through methodologies, such as Team Software Process (TSP) and Personal Software Process (PSP) to improve performance and production of virtually defect free software without overrunning budget or schedules [Humphrey, 2005; Davis &

Mullaney, 2003; Rombach *et al.*, 2008]. In PSP, productivity is measured as "size" units per hour (LOC per hour). It is also measured per process phase, because it is valuable to know which phase are the problematic ones and require more attention. The important part of TSP and PSP is the measurement framework consisting of four measures: effort, schedule, size and defects [Pomeroy-Huff *et al.*, 2009]. It has methods for improving process, project management and quality management. Software development processes that intensively use metrics and quantitive methods can generate vast amount of data to be analyzed [Burton & Humphrey, 2006]. This data aids in identification of performance problems, determination of root causes and generation of improvement actions.

PSP has a suggested measure called Process Quality Index (PQI) which takes five different components into account: design to code ratio, design review to design ratio, code review to code ratio, defect density in compilation and defect density in unit test. Design to code ratio is design quality in time ratio. Design review to design ratio is design review quality in time ratio. Code review to code ratio is code review quality in time ratio. Defect density in compilation is the code quality in defects per KLOC. Defect density in unit test is the software quality in defects per KLOC. Raza & Faria [2014] report that too small design to code ratio usually relates to a lack of thoroughness or even total absence of the design artifacts produced, such as important design views or coverage of requirements.

6.2.1 Quality Assurance Program

Quality assurance (QA) consists of various techniques and plans for management to promote high quality process and products. Quality and risks are analyzed by reviewing plans, procedures, software (requirements, design, documentation), schedules and reports. There are several approaches to improve software quality, such as Capability Maturity Model (CMM) [Jalote, 2000], total quality management (TQM) and Six Sigma [Pyzdek & Keller, 2003]. Different approaches aid quality management in given situations by identifying available improvement opportunities. Sun [2011] notes that software quality should be controlled in every step through the life-cycle of a project all the way to the maintenance phase, because each project phase affects the final software quality. For example, a defect that is identified in testing phase could have been occurred much earlier

in the requirement specification phase. Hence, quality control promotes correct requirement specification.

The main benefit of quality control is to establish clear and formal quality delivery, but as a downside it involves high bureaucracy and lots of documentation [Wagner, 2013]. However, the decision to save project budget on quality control causes the software quality to decrease. Gilb [2000] mentions feedback as single and most powerful principle to drive quality and promote success. He also points out that methods using feedback provide good project control and practical information. Jones [2008] presented five steps to software quality control for medium and large enterprises:

1. Software quality metric program establishment.
2. Tangible executive software performance goal establishment.
3. Software quality assurance establishment (defect prevention and removal).
4. Leading-edge corporate culture development for formal quality programs.
5. Software strength and weakness identification.

Software industry has lot of uncertainty and successful projects require good management of risks that are involved. Shihab *et al.* [2012] sum that risk management has proven its successful functionality and raised interest among researchers and industry [Freimut *et al.*, 2001; Miccolis *et al.*, 2001]. The interest has made the industry and related researchers more active in risk management.

In software bug prediction future bug appearance is predicted based on code and historical metrics [Shihab *et al.*, 2012]. Although statistical fault prediction models outperform human estimation, software industry still relies on subjective predictions made by human experts [Tomaszewski, 2006].

6.2.2 Quality Cost Control

Companies often promote quality as the central customer value and describe it as an critical factor for success in achieving competitiveness [Schiffauerova & Thomson, 2006]. However, organizations do not usually have a realistic knowledge on their costs, for example, how much profits they are losing because of low internal quality. In addition, Tatikonda & Tatikonda [1996] and Morse [1993] state that only few managers measure the results of quality improvement activities.

The reduction of costs that improve quality, is only possible if the costs are identified and measured. This means that the cost of quality (CoQ) management is essential. CoQ is usually understood as the sum of conformance (poor quality prevention costs) and non-conformance (costs caused by poor quality) costs [Machowski & Dale, 1998]. The goal of a CoQ model is to find a level of quality that minimizes the total cost of quality. CoQ models are useful in understanding the economic trade-offs involved in delivering good quality software. Many examples confirm that quality improvement and cost measurement reduces quality costs considerably [Schiffauerova & Thomson, 2006]. Schiffauerova & Thomson [2006] list that detailed metrics in CoQ include the following:

- Asset and material costs.
- Preventive labor costs.
- Appraisal labor costs.
- Cost of defects per hundred pieces.
- Late delivery costs.
- Percentage of repeat sales.
- Time between service calls.
- Number of non-conforming calls.
- Number of complaints received.

There are various cost models for software quality, such as P-A-F (prevention-appraisal-failure) [Slaughter *et al.*, 1998; Knox, 1993; Krasner, 1998], Crosby's model [Crosby, 1979], process cost model [Ross, 1977] and ABC (activity based costing) model [Cooper & Kaplan, 1988]. Most CoQ models are based on the P-A-F classification [Plunkett & Dale, 1987; Machowski & Dale, 1998; Sandoval-Chávez & Beruvides, 1998]. For decisions related to costs, there is COCOMO (Constructive Cost Model) that is a popular cost estimation model originally published by Boehm [1981]. It was originally published as COCOMO 81, but it was later revised by Boehm *et al.* [1995] and published as COCOMO II. COCOMO II is used in decision making concerning, say, project budgets and schedules or tradeoffs among software cost, features, quality and schedule.

CoQ models are designed to prevent poor quality, but they are rarely used [Krasner, 1998], because they do not have much appreciation in organizations. Since formal methods are rarely used, other approaches are considered for cost measurements.

6.2.3 Testing

In programming, the majority of time is spent debugging defects instead of implementing new features [Fowler *et al.*, 1999]. Continuous testing is an effective way to manage bugs by giving useful information for debugging procedures. Software testing is a verification process in which the behavior of the program is tested with a set of test cases against the expected behavior [McConnell, 2004]. Testing is an essential part of development to reduce unnecessary risks and costs related to quality.

Unit testing means that software components are tested to meet the technical requirements. Tests are written before the implementation to correspond with the required functionality. When a test passes, it indicates that the unit works as intended, because the requirements that the unit test defines are filled [Fowler *et al.*, 1999]. Manual testing is inefficient, since time spent manually evaluating the source code might take multiple times more effort compared to automated testing. Automated testing either approves or rejects the tested component source code and gives only a status statement as an output to programmer.

Since unit testing evaluates the internal quality, functional testing is used in QA to test the external quality of the software. Functional testing is performed to assure that the quality that users experience is on appropriate level. These tests are not meant to be used during active development when agile methods and high productivity are required.

6.2.4 Technical Reviews

To enable any process to work adequately, active communication between and within stakeholders and customer/user feedback are central elements. In software development, review meetings, such as source code reviews, are a good way to sustain comprehensive technical understanding between development team members [Fowler *et al.*, 1999]. Different team members complement each other's understanding concerning technical implementations and visions because of their varying specializations.

According to Kemerer & Paulk [2009], the recommended rate for individual re-

views are 200 or less LOC per hour. This helps to identify two-thirds of the defects in design reviews and more than half in code reviews [Raza & Faria, 2014]. Reviews help in issue inspection from multiple perspectives and help avoiding design and implementation flaws.

6.2.5 People Quality

Boehm [1988] reports that the performance of people are a key matter in productivity improvement. Hence, management, staffing, incentives and working environment are significant factors in productivity. Pragmatic programmer is an easy adopter and a fast adapter, which means that one has an instinct for technologies and techniques [Hunt & Thomas, 1999]. Being pragmatic helps also being experimental and getting experience from multiple areas to increase confidence and adaptability. With confidence it is easier to be critical thinker and face challenges. Also, being realistic aids to avoid absurd goals that cause unnecessary pressure for development and decrease productivity. Learning new technologies and methods is important for managers and developers to maintain good level of know-how and ability to adapt to environmental changes.

6.3 Improving Internal Quality

This section continues to answer the research question *"How to promote productive development?"* and presents source code related elements. The main focus in development is often required to be in the customer-related work. Good design, comprehensibility and reuse are top issues now and in the future.

6.3.1 Software Architecture

Architecture in a physical built object means "unifying or coherent form or structure" [Dictionary, 2014]. In software development the "building materials" consist of programming language constructs and the operating environment they are executed in. In addition, there are abstract concepts which represent certain building components and their arrangement. The ability to read and understand the source code of a software is comparable to reading a description of a building

instead of viewing the building itself [Baragry & Reed, 1998]. In other words, software does not have a physical representation to be viewed or touched.

Software development starts with conceptual model construction of the implemented system. The model includes structural and logical arrangement of abstract high-level concepts. Conceptual model is created to aid the vision of the solution and how it should work. The model shows structural concepts and relationships that are needed in the actual implementation. This creation process involves many problems, such as requirement definition, transformation to implementation and the operation as an explanatory theory [Baragry & Reed, 1998]. In addition, architects have differing levels of education, known design methods and experience which have an effect on how architectural concepts are designed. Generally, there is no design that would be unambiguously right because some concepts suit better to certain situations than others.

The term "architecture" is over-used in software engineering and its meaning is scattered. Clements & Northrop [1996] describe reasons why practitioner community has not been able to define software architecture requirements in a consensus. Methodological biases in architecture are found to be a common problem. They also acknowledge that the whole field is still quite new, and the study of software architecture is following the practice instead of leading it. Poor foundations, such as undefined and ambiguous terms, also increase confusion.

Stroggylos & Spinellis [2007] point out that the software design is the most influential factor for software quality, because it could evolve with fewer effort and cost. The success of software development depends heavily on the architectural design, because of the increasing size and complexity of software [Chen *et al.*, 2010]. Architectural styles and patterns are key elements to achieve better designs. The styles and patterns exploit the existing system design knowledge to provide simplified software designs that make design reuse easier. Software architecture also evolves over time. For example, variations in requirements, technology, environment or distribution cause architectural changes. Static evolution happens in the software specification phase and dynamic evolution happens in software runtime. The dynamic evolution involves high risk, because of its complexity and difficulty [Chen *et al.*, 2010].

6.3.2 Software Maintainability

Programming, debugging and testing covers 30–50% of a software project [McConnell, 2004], so high internal quality is important for optimal maintainability. Maintenance might exceed 60% of the development costs being the most laborious phase in development [Bell *et al.*, 1992]. Developers also spend more effort on software debugging than new feature development [Fowler *et al.*, 1999]. Hence, the quality of source code becomes even more significant factor to prevent budget overruns and project failures. Software architecture is often required to be flexible to promote reuse and be able to respond to changing requirements. Flexibility makes changes easier to perform, but the maintenance becomes more expensive, because of increased code base and complexity.

Old technologies, poor architecture design or other legacy systems could prevent system scaling, new features or modern development practices. Software rewrite could be the only option to regain appropriate internal quality. Rewrite usually comes into question, when refactoring cannot be performed incrementally in smaller parts. When a software is rewritten, it is important to keep the internal quality high to prevent yet another redesign demand in the future [Martin, 2009].

Software comprehension is a relevant part of maintenance [Roehm *et al.*, 2012]. On average the time needed for comprehension is half of the time developers spend on maintenance [Fjeldstad & Hamlen, 1983]. However, Roehm *et al.* [2012] report developers usually avoid comprehension to save time and mental effort. Source code is often duplicated and modified to their needs to complete a task at hand. Developer experience and conformed standards are found to aid in new source code familiarization and modern comprehension tools are rarely utilized [Roehm *et al.*, 2012].

6.3.3 Development Frameworks and Libraries

As Boehm [1988] states, software productivity can be improved through software component reuse, because it decreases the size of code base. There are vast amount of different development frameworks and libraries available. Available frameworks and libraries are utilized in development to promote source code reuse, standardization and quality. Reusing existing components enables soft-

ware development to concentrate on the actual project implementation, when the framework has already been established and can just be used.

Available development technologies affect productivity, hence technological changes can be used to improve productivity [Sommerville, 2010]. However, productivity usually decreases when technological changes are executed in a project, but is improved above the original level when the new technologies are in operation and mastered.

Framework development requires significant effort in a software project when it is implemented from scratch. Framework developers usually promote good architectural decisions and designs to be able to provide developers the right tools in the right way. Whether an organization decides to refactor poor legacy architecture or build a new framework to improve current internal quality, both will require significant amount of resources. Framework development decreases development productivity, because the development effort increases without increasing the actual software product size. Developer competence affects the most on source code delivery rate, but lack of common source code libraries significantly decrease the delivered functionality [Tomaszewski, 2006]. The benefits of using existing third-party frameworks include the following:

- Organizations can hire developers that are familiar with the used framework.
- Resources required to develop, maintain and test a good quality framework are saved, when the work is outsourced to the framework developer.
- Risks and issues are outsourced to the framework developer.
- Extensive support due to large user community.
- Standardizations (code and file structures, naming conventions and other foundational practices), security and design quality are evolved through extensive framework developer and user community contribution.

6.3.4 Development Tools

There are multiple tools available for development to improve management and productivity. Integrated development environments (IDE) offer tools for syntax checking, automated refactoring, testing, version control and so on. Microsoft Vi-

sual Studio[1], Jetbrains IDEA[2] and Oracle NetBeans[3] are few examples of these. There are multiple development-related tools available online these days, for example, project management tools (e.g., Atlassian JIRA[4], Microsoft Project[5] or JetBrains YouTrack[6]), software hosting and version control (e.g., GitHub[7], SourceForge[8] or Google Code[9]) or design diagramming (e.g., Lucidchart[10] or Gliffy[11]).

6.4 Managing Technical Debt for Sustainability

In this section, the research question *"How to sustain productive development?"* is answered through technical debt management. Technical debt management is suggested as the solution to sustain productivity in software development through high internal quality administration.

Technical debt is managed to provide a good foundation for project trajectory. Management practices include examination and establishment of debt estimation model, bad programming practices, debt tolerance levels and debt monitoring [Letouzey & Ilkiewicz, 2012]. When technical debt management aims to aid decision-making, the debt measurement must be comparable and monetizable [Brown *et al.*, 2010]. If technical debt is not measured, managers need to make decisions based on their experience [Guo *et al.*, 2011].

Krishna & Basu [2012] divide different debt reduction techniques into three sections: identification, classification and reduction. Firstly, insufficient documen-

[1] `http://www.visualstudio.com`

[2] `https://www.jetbrains.com/idea`

[3] `https://www.netbeans.org`

[4] `https://www.atlassian.com/software/jira`

[5] `http://office.microsoft.com/en-001/project`

[6] `https://www.jetbrains.com/youtrack`

[7] `https://github.com`

[8] `http://sourceforge.net`

[9] `https://code.google.com`

[10] `https://www.lucidchart.com`

[11] `http://www.gliffy.com`

tation or code quality and coverage is *identified*. Then the identified debt is *classified* as knowingly or unknowingly, short-term or long-term, reckless, strategic or non-strategic and four grades of debt[12]. Lastly, the technical debt is *reduced* by refactoring, TDD, source code reviews and audits, pair programming, continuous integration, best practices and standards, and evolutionary design. TDD is considered very beneficial for ongoing technical debt reduction according to TDD experiments [Krishna, 2011], but it is reported that effective remediation methods are still missing. Krishna & Basu [2012] identify thirteen steps for debt management:

1. Identify and define a "living budget" that is the minimum production effort (includes estimation and planning of code reviews and refactorings) that has to be made to meet the deadlines.
2. Make sure to have time to understand why poor quality code is poor.
3. Identify the need to over-anticipate and eliminate it to prevent unnecessarily complex code design.
4. Base designs on knowledge instead of assumptions.
5. Communicate and exercise pair programming.
6. Avoid the urge to please others — design the best way you can.
7. Follow standards and best practices.
8. Refactor one part at a time.
9. Utilize spare work time for debt reduction.
10. Keep yourself organized: estimate, monitor, empower others and prioritize.
11. Increase productivity and measure it in quality, not in quantity.
12. Learn and apply different refactoring techniques.
13. Keep learning constantly.

For technical debt management, there is a method called SQALE (Software Quality Assessment Based on Life-cycle Expectations) [Letouzey & Ilkiewicz, 2012]. SQALE provides guidelines for technical debt estimation and refactoring planning. To be able to estimate technical debt, organization is required to define programming guidelines to represent a quality model, which is a contract in the development team. The model includes requirements concerning architecture and structure, implementation, naming conventions, legacy code management rules and presentation. Also, there should be training for the non-technical stakehold-

[12] `http://madebymany.com/blog/the-four-grades-of-technical-debt`

ers to share knowledge concerning technical debt and SQALE.

In SQALE, each software requirement must associate with remediation function, which turns the amount of noncompliances to technical debt remediation cost [Letouzey & Ilkiewicz, 2012]. The remediation costs for all the noncompliances are the actual technical debt. If technical debt is accepted, non-remediation costs are incurred. In SQALE the debt is called SQALE quality index (SQI). The SQALE method has eight quality characteristics: testability, reliability, changeability, efficiency, security, maintainability, portability and reusability. Testability is the base for all the other characteristics, because untestable code makes quality assurance very difficult.

6.4.1 Debt Visualization

Monitoring any health condition is beneficial and in many situations the critical thing to do. Monitoring software health is equally important to functional requirement conformance or meeting development milestone deadlines. Furthermore, technical debt might be misunderstood collectively, cause delays to the project and compromise the software quality. When the state of internal quality is monitored, the organization can react to technical debt. Technical debt is hard to be removed completely, but it is not advised to do so. In contrast, a vast amount of debt can lead to failure or increased costs through additional effort and rework [Krishna & Basu, 2012]. However, technical debt's incurring costs (interest cost) are hard to be measured or estimated [Zazworka *et al.*, 2013].

To visualize technical debt, dashboards are a good way to make the information available. Dashboards also enable transparent communication within the organization. Power [2013] explains that technical debt often projects to feature velocity in a project that is seen by business stakeholders. The problem is that they might not know the real reason for the decrease and do not see where developers are actually investing their time — which might be the technical debt. When debt is being tracked continuously, development team is able to compare planned effort investments to actual investments. Power [2013] reminds that even entire release cycle could be spent solely on debt reduction. Technical debt measurement should be a highly automated process to avoid additional employee workload.

Snipes *et al.* [2012] make a rhetoric question: "Is a policy that drives the product towards zero debt the best for all stakeholders?" Ramasubbu & Kemerer [2013] report that the tradeoffs in technical debt are not fully studied to reveal the facts concerning the benefits and disadvantages of having technical debt. As a key obstacle they suggested to model the evolutionary nature of the technical debt accumulation. This would take into account the benefits and costs of technical debt that is gained through the life-cycle of a software system.

6.4.2 Decision Making

Business competition decreases the value of existing software. Hence, companies have to add more value to their software products by developing new features. However, when an organization wants to improve the business value through software quality, there are many details it has to investigate. These details include current quality status and costs related to quality. After various details are investigated, the organization can evaluate production costs, benchmarks and standards. Also, economic trade-offs and poor quality costs become visible and assist in future decision making [Krasner, 1998]. Therefore, software organizations need to understand how much technical debt they have and how much debt reduction they can afford [Power, 2013]. Snipes *et al.* [2012] list the following factors for decision-making concerning deferral of defects in order of importance:

1. Defect severity.
2. Existence of a workaround.
3. Fix urgency specified by the customer.
4. Implementation effort.
5. Risk of fix proposal.
6. Testing scope.

Fowler *et al.* [1999] state that there are no exact guidelines for the decision making concerning the verdict between refactoring current software and a fresh start. However, the decision is easier when refactoring software to separate components, since the problem area is narrowed down and different parts can be refactored separately. In general, a component or a software could be rebuilt from scratch if the development has become unsustainable or if the current software cannot be transformed to meet new visions and objectives. Reasons for unsustainable

development could be, for example, high software defect count or foundational architecture design flaws. Making changes to already malfunctioning code is always risky, because the modifications could cause even more defects. The majority of source code should always be defect-free [Fowler *et al.*, 1999].

6.4.3 Debt Remediation

Developers play an important role in technical debt remediation as being key stakeholders in software projects [Krishna & Basu, 2012]. The most experienced and skilled developers are not usually interested in dealing with technical debt, because they tend to promote high quality practices in the first place. When technical debt is addressed and reduced, there is always a risk that it results in more issues that there was before the refactoring. This is why source code modification is unattractive in general, because modifications involve high risks compared to new implementations [Tomaszewski, 2006].

Refactoring is a key strategy to improve quality, enable testing and decrease defects [Ktata & Lévesque, 2010; McConnell, 2004; Fowler *et al.*, 1999; Mens & Tourwé, 2004]. The cardinal rule in software evolution is that evolution should improve the source code quality and the key strategy to achieve this is to refactor continuously [McConnell, 2004]. Software architects' are responsible to provide guidance for refactoring decision-making, re-engineering and rewriting [Buschmann, 2011]. Pragmatic architects often make their decisions based on business questions to be able to evaluate when and how to reduce technical debt through refactoring. When every software requires refactoring because of iterative development, the easiest way to refactor and save resources is to integrate refactoring in the development process and do it systematically [Veerraju *et al.*, 2010].

7 Conclusion

7.1 Summary

Modern information society cannot function without reliable software. However, software development projects are very complex and have high risks. Inaccurate specification/estimation, budget overruns, delivery delays and lack of management are common issues to struggle with. Roughly 84% of software projects fail and about half of them has reduced functionality while around one third are cancelled before completion. The general issue in software development is that the internal software quality issues are often neglected, although programming, debugging and testing forms 30–50% of a software project. The correctness and the quality of software form the success.

There are at least two foundational requirements for any project — project success and staying on budget. Comprehensive scope is an essential requirement to promote accurate estimation. One major problem with estimation in software projects is the lack of public data in the industry. The root problems in cost estimation include missing required variable information and lack of estimation tool experience.

The end-product quality of any software project is heavily connected to quality measurements. However, lack of budget and monitoring for quality is a common issue in small organizations. The major issue for the lack of quality measurement might be the expensiveness of quality control. However, poor quality accumulates costs as well, which can rise even higher than the costs of quality promotion. Since the quality of people is a significant factor in productivity, technological training becomes considerable option for small organizations with limited resources.

High internal quality means readable and understandable source code and architecture, that improves software maintainability (analyzability, modifiability, modularity, reusability, testability). High maintainability enables developers to do their work appropriately — to be productive. Development productivity is mostly affected by the developer competence, but software requirement complexity, training, available technologies and tools, and the overall process quality also have significant impact on it.

Good internal software structure is expected to provide good external quality as well [Fenton & Pfleeger, 1998]. Internal software quality should be improved, since low internal quality score results in external quality score that is below the average at maximum [Stavrinoudis & Xenos, 2008]. High quality prevents inadequate functionality or validation that cause significant problems to software systems due to generation of unreliable data.

Internal software quality is decreased by technical debt which is the cost of fixing structural problems in production code. Gartner[1] reports the IT (technical) debt to be $500 milliard in 2010 and potentially rising to $1 billion by 2015. Unintentional debt is a challenge in software development and is mainly accumulated by incompetent individuals. Technology training can be used to decrease the debt accumulation related to incompetence. Training enables developers to follow standards and design better architecture for solid software foundation. This is beneficial for cost minimization because re-engineering poor architecture later in a project incurs high costs.

Intentional debt is accumulated through postponed change requests, refactoring and debugging, hence intentional debt should be avoided by appropriate prioritization — by choosing the software health over new features. This requires that the business stakeholders understand the technical aspects and vice versa, thus close collaboration and active communication are foundational requirements. The health of the software, as in internal quality, is important since the majority of development is defect fixing and maintenance. The earlier and easier any defects are fixed in software, the less they accumulate costs to business during a project. Improvements in defect detection and removal are likely to have a significant impact on development productivity, as they are almost half of the software project costs.

Organization has to know the state of technical debt and how much reduction it can afford. Technical debt is reduced, e.g., by refactoring, test-driven development, reviews and audits, standards and evolutionary design. As a checklist for debt management, Krishna & Basu [2012] identify thirteen steps to go through. Technical debt should be thought and inspected the same way as financial debt and make it more visible and meaningful for management. Debt has to be identi-

[1] `http://www.gartner.com/newsroom/id/1439513`

fied, measured and visualized, so it can be remediated by refactoring in software projects.

The refactoring benefits and costs are usually not quantified, so it is difficult to justify it to be mandatory in general. Software comprehension being half the time spent on maintenance, internal quality becomes significant factor in additional cost minimization. High internal quality means better productivity, and better productivity means lower costs. Thus, refactoring should be a major concern in development, since internal quality directly affects the business. To sustain productive development and prevent project failures, continuous refactoring to reduce technical debt is required.

7.2 Discussion

7.2.1 Technical Debt and Small Organizations

Software startup-companies often struggle with the lack of resources. So, how they should manage technical debt? When software quality is generated from the process and people involved in it, there is a question which one to invest in. Process improvement can be expensive because of the extensive quality control activities. However, certain parts of good quality process could be adopted as beneficial but reasonably cheap, such as internal quality measurement and visualization, and defect prevention with appropriate testing process. These two decreases costs by making the internal quality issues visible for the development and minimizing the defects.

The other option would be to invest in people quality. This means that more competent and expensive developers are hired in the first place, or less skilled programmers are trained to follow standards and use advanced technologies. Investing in people might be a best option for starting software companies to be able to kickstart their development in high velocity and maintain it.

However, there might be a third option to increase the project success probability much higher and to handle the high quality internally. The third option is to keep the project size small and make the requirements as simple and manageable as possible. In other words, the product and the business model have to be planned

carefully. This way the risks and the need to cut corners are decreased, when "there is time to concentrate on every quality".

Software that is built short-term benefits in mind might also be a good journey of lessons for the development team to brainstorm how the business visions can be transformed into working foundations. When the development paths are examined the issues and good decisions are learned. This knowledge could be used in the next software version, which would be rebuilt from scratch. However, this might be more difficult than expected. The initial release cannot be too large to enable maintainable and simple re-engineering.

7.2.2 Is the Technical Debt Phenomenon Only a Lack of Competence?

The reasons for debt (incompetence and inadequate process) culminate in one major factor — incompetence of individuals. Hence, the incompetence seems to be the root cause for debt, does this mean that technical debt is just mistakes instead of any "intentional decisions" to gain any advantages on anything. Is the inaccurate estimation the root cause for the whole phenomenon of technical debt, or is it just a collection of mistakes? In other words, is the majority of projects unmanageable in the first place? This could mean that when software requirements are engineered, there could be foundational mistakes to make, such as:

- Specified project size is too large to be manageable.
- Absurd initial deadline which is not rejected.
- Software requirements are too complex for a single project.
- Lack of competence in the project team to be able to meet objectives.
- Inaccurate estimation on anything.

The issue with technical debt becomes a bit more absurd, when it is compared to another field of industry. As an analog comparison, would a building constructor cut corners to meet external requirements and compromise the safety or maintainability of the building? As a conclusion, it would make a lot of sense to integrate the quality management and sustainable productivity improvement as part of software development. Thus, this area needs more research and a development of a framework to address the need for easy and affordable management.

7.3 Future Research

As continuous increase in complexity is an issue for future software development and management, the management of complexity should be researched. The relation between source code complexity and productivity can be researched to estimate how the increasing complexity affects the development productivity. Software reuse and its innovation can be a huge topic in the future to decrease risks and consequences related to complex and large software modification.

Quality is tradable attribute [Fowler, 2011] and it depends on the context of the project [Krasner, 1998; Fenton & Pfleeger, 1998], so it is difficult to justify its essentiality. Also, the benefits or costs of refactoring are not usually quantified, hence the key points for justification are required. This topic requires further research and a generation of a model that could quantify the importance of quality to aid decision-making or prove the internal quality and productivity improvement as mandatory activity in software projects.

References

[Austin, 2001] Robert D Austin. The effects of time pressure on quality in software development: An agency model. *Information Systems Research*, 12(2):195–207, 2001.

[Banker & Slaughter, 2000] Rajiv D Banker & Sandra A Slaughter. The moderating effects of structure on volatility and complexity in software enhancement. *Information Systems Research*, 11(3):219–240, 2000.

[Baragry & Reed, 1998] Jason Baragry & Karl Reed. Why is it so hard to define software architecture? In *Proceedings of the Asia Pacific Software Engineering Conference*, pages 28–36. IEEE, 1998.

[Bell *et al.*, 1992] Doug Bell, Ian Morrey, & John Pugh. *Software Engineering: A Programming Approach*. Prentice Hall, 1992.

[Boehm *et al.*, 1976] Barry W Boehm, John R Brown, & Mlity Lipow. Quantitative evaluation of software quality. In *Proceedings of the 2nd International Conference on Software Engineering*, pages 592–605. IEEE Computer Society Press, 1976.

[Boehm *et al.*, 1978] Barry W Boehm, John R Brown, & Hans Kaspar. Characteristics of software quality. *TRW Series of Software Technology*, 1978.

[Boehm *et al.*, 1995] Barry Boehm, Chris Abts, B Clark, & S Devnani-Chulani. Cocomo ii. `http://csse.usc.edu/csse/research/COCOMOII/cocomo_main.html`, 1995. [Online; accessed November 11., 2014].

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

[Boehm, 1988] Barry W. Boehm. Understanding and controlling software costs. *Journal of Parametrics*, 8(1):32–68, 1988.

[Boehm, 2006] Barry Boehm. Some future trends and implications for systems and software engineering processes. *Systems Engineering*, 9(1):1–19, 2006.

[Bohnet & Döllner, 2011] Johannes Bohnet & Jürgen Döllner. Monitoring code quality and development activity by software maps. In *Proceedings of the 2nd Workshop on Managing Technical Debt*, pages 9–16. ACM, 2011.

[Brown *et al.*, 2010] Nanette Brown, Yuanfang Cai, Yuepu Guo, Rick Kazman, Miryung Kim, Philippe Kruchten, Erin Lim, Alan MacCormack, Robert Nord, Ipek Ozkaya, et al. Managing technical debt in software-reliant systems. In *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research*, pages 47–52. ACM, 2010.

[Burton & Humphrey, 2006] Dan Burton & Watts Humphrey. Mining PSP data. In *TSP Symposium*, 2006.

[Buschmann, 2011] Frank Buschmann. To pay or not to pay technical debt. *IEEE Software*, 28(6):29–31, 2011.

[Chappell, 2014] David Chappell. The three aspects of software quality: Functional, structural, and process. `http://www.davidchappell.com/writing/white_papers/The_Three_Aspects_of_Software_Quality_v1.0-Chappell.pdf`, 2014. [Online; accessed June 4., 2014].

[Chen *et al.*, 2010] Yao Chen, Xiaoqing Li, Lingyun Yi, Dayong Liu, Liu Tang, & Hongli Yang. A ten-year survey of software architecture. In *Proceedings of the 2010 IEEE International Conference on Software Engineering and Service Sciences*, pages 729–733, 2010.

[Chrissis *et al.*, 2003] Mary Beth Chrissis, Mike Konrad, & Sandy Shrum. *CMMI Guidlines for Process Integration and Product Improvement*. Addison-Wesley, 2003.

[Clements & Northrop, 1996] Paul C Clements & Linda M Northrop. Software architecture: An executive overview. Technical report, DTIC Document, 1996.

[Cooper & Kaplan, 1988] Robin Cooper & Robert S Kaplan. Measure costs right: make the right decisions. *Harvard Business Review*, 66(5):96–103, 1988.

[Crosby, 1979] Philip B Crosby. *Quality is Free: The Art of Making Quality Certain*. McGraw-Hill, 1979.

[Cunningham, 1992] Ward Cunningham. The wycash portfolio management system. *ACM SIGPLAN OOPS Messenger*, 4(2):29–30, 1992.

[Curtis *et al.*, 2012] Bill Curtis, Jay Sappidi, & Alexandra Szynkarski. Estimating the size, cost, and types of technical debt. In *Proceedings of the 3rd International Workshop on Managing Technical Debt*, pages 49–53. IEEE, 2012.

[Davis & Mullaney, 2003] Noopur Davis & Julia L Mullaney. The team software process (tsp) in practice: A summary of recent results. *Software Engineering Institute, Carnigie Mellon University*, 2003.

[Diaz & Sligo, 1997] Michael Diaz & Joseph Sligo. How software process improvement helped motorola. *IEEE Software*, 14(5):75–81, 1997.

[Dictionary, 2014] Merriam Webster's Collegiate Dictionary. Merriam-Webster Dictionary. `http://www.merriam-webster.com/netdict.htm`, 2014. [Online, accessed July 23., 2014].

[Dromey, 1995] R. Geoff Dromey. A model for software product quality. *IEEE Transactions on Software Engineering*, 21(2):146–162, 1995.

[Dromey, 1996] R Geoff Dromey. Cornering the chimera. *IEEE Software*, 13(1):33–43, 1996.

[Dumas & Redish, 1999] Joseph S Dumas & Janice Redish. *A Practical Guide to Usability Testing*. Intellect Books, 1999.

[Ergin, 2013] Lemi Orhan Ergin. Technical debt: Do not underestimate the danger. `http://www.slideshare.net/lemiorhan/technical-debt-do-not-underestimate-the-danger`, 2013. [Online; accessed August 25., 2014].

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

[Fenton *et al.*, 2004] Norman Fenton, William Marsh, Martin Neil, Patrick Cates, Simon Forey, & Manesh Tailor. Making resource decisions for software projects.

In *Proceedings of the 26th International Conference on Software Engineering*, pages 397–406. IEEE, 2004.

[Fenton *et al.*, 2008] Norman Fenton, Martin Neil, William Marsh, Peter Hearty, Łukasz Radliński, & Paul Krause. On the effectiveness of early life cycle defect prediction with bayesian nets. *Empirical Software Engineering*, 13(5):499–537, 2008.

[Fjeldstad & Hamlen, 1983] Richard K Fjeldstad & William T Hamlen. Application program maintenance study: Report to our respondents. *Proceedings Guide*, 48, 1983.

[Fowler *et al.*, 1999] Martin Fowler, Kent Beck, John Brant, William Opdyke, & Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[Fowler, 2003] Martin Fowler. Technical debt. `http://martinfowler.com/bliki/TechnicalDebt.html`, 2003. [Online; accessed July 1., 2014].

[Fowler, 2011] Martin Fowler. Tradable quality hypothesis. `http://martinfowler.com/bliki/TradableQualityHypothesis.html`, February 2011. [Online; accessed August 19., 2014].

[Fox & Frakes, 1997] Christopher Fox & William Frakes. The quality approach: is it delivering? *Communications of the ACM*, 40(6):24–29, 1997.

[Freimut *et al.*, 2001] Bernd Freimut, Susanne Hartkopf, Peter Kaiser, Jyrki Kontio, & Werner Kobitzsch. An industrial case study of implementing software risk management. *ACM SIGSOFT Software Engineering Notes*, 26(5):277–287, 2001.

[Gaffney, 1984] John E Gaffney. Estimating the number of faults in code. *IEEE Transactions on Software Engineering*, (4):459–464, 1984.

[Garvin, 1984] David A. Garvin. What does "product quality" really mean? *MIT Sloan Management Review*, 26(1):25–43, October 1984.

[Gilb, 2000] Tom Gilb. The ten most powerful principles for quality in (software and) software organizations for dependable systems. *Computer Safety, Reliability and Security*, pages 1–13, 2000.

[Glass, 2001] Robert L Glass. Frequently forgotten fundamental facts about software engineering. *IEEE Software*, 18(3):112–112, 2001.

[Grady, 1992] Robert B Grady. *Practical Software Metrics for Project Management and Process Improvement*. Prentice-Hall, 1992.

[Guo *et al.*, 2011] Yuepu Guo, Carolyn Seaman, Rebeka Gomes, Antonio Cavalcanti, Graziela Tonin, Fabio QB Da Silva, André Luis M Santos, & Clauirton Siebra. Tracking technical debt — an exploratory case study. In *Proceedings of the 27th IEEE International Conference on Software Maintenance*, pages 528–531. IEEE, 2011.

[Harter & Slaughter, 2000] Donald E Harter & Sandra A Slaughter. Process maturity and software quality: a field study. In *Proceedings of the 21st International Conference on Information Systems*, pages 407–411. Association for Information Systems, 2000.

[Higgs, 2011] James Higgs. The four grades of technical debt. `http://madebymany.com/blog/the-four-grades-of-technical-debt`, 2011. [Online; accessed July 1., 2014].

[Hihn & Habib-agahi, 1991] Jairus Hihn & Hamid Habib-agahi. Cost estimation of software intensive projects: a survey of current practices. In *Proceedings of the 13th International Conference on Software Engineering*, pages 276–287. IEEE Computer Society Press, 1991.

[Humphrey, 2005] Watts S Humphrey. *PSP (sm): A Self-Improvement Process for Software Engineers*. Addison-Wesley Professional, 2005.

[Hunt & Thomas, 1999] Andrew Hunt & David Thomas. *The Pragmatic Programmer*. Addison-Wesley, 1999.

[IEEE, 1993] IEEE. Standard for software productivity metrics. IEEE 1045-1992, IEEE Computer Society, 1993.

[ISO, 1997] ISO. Quality management and quality assurance standards – part 3: Guidelines for the application of iso 9001:1994 to the development, supply, installation and maintenance of computer software. ISO 9000-3:1997, International Organization for Standardization, 1997.

[ISO, 2000] ISO. Quality management systems – requirements. ISO/IEC 9001:2000, International Organization for Standardization, 2000.

[ISO, 2001] ISO. Software engineering – product quality – part 1: Quality model. ISO/IEC 9126-1:2001, International Organization for Standardization, 2001.

[ISO, 2005] ISO. Quality management systems – fundamentals and vocabulary. ISO 9000:2005, International Organization for Standardization, 2005.

[ISO, 2010] ISO. Systems and software engineering – vocabulary. ISO/IEC/IEEE 24765:2010, International Organization for Standardization, 2010.

[ISO, 2011] ISO. Systems and software engineering – systems and software quality requirements and evaluation (square) – system and software quality models. ISO/IEC 25010:2011, International Organization for Standardization, 2011.

[Jalote, 2000] Pankaj Jalote. *CMM in Practice: Processes for Executing Software Projects at Infosys*. Addison-Wesley Professional, 2000.

[Jones, 2008] Capers Jones. *Applied Software Measurement: Global Analysis of Productivity and Quality*, volume 3. McGraw-Hill New York, 2008.

[Jørgensen, 1999] Magne Jørgensen. Software quality measurement. *Advances in Engineering Software*, 30(12):907–912, 1999.

[Jørgensen, 2004] Magne Jørgensen. A review of studies on expert estimation of software development effort. *Journal of Systems and Software*, 70(1):37–60, 2004.

[Kemerer & Paulk, 2009] Chris F Kemerer & Mark C Paulk. The impact of design and code reviews on software quality: An empirical study based on psp data. *IEEE Transactions on Software Engineering*, 35(4):534–550, 2009.

[Kerievsky, 2004] Joshua Kerievsky. *Refactoring to Patterns*. Addison-Wesley, 2004.

[Kernighan & Plauger, 1978] Brian W. Kernighan & Phillip J. Plauger. *The Elements of Programming Style*. McGraw-Hill, 2nd edition, 1978.

[Klinger *et al.*, 2011] Tim Klinger, Peri Tarr, Patrick Wagstrom, & Clay Williams. An enterprise perspective on technical debt. In *Proceedings of the 2nd Workshop on Managing Technical Debt*, pages 35–38. ACM, 2011.

[Knox, 1993] Stephen T. Knox. Modeling the cost of software quality. *Digital Technology Journal*, 5(4):9–17, 1993.

[Krasner, 1998] Herb Krasner. Using the cost of quality approach for software. *Crosstalk*, 11:6–11, November 1998.

[Krishna & Basu, 2012] V Krishna & A Basu. Minimizing technical debt: Developer's viewpoint. In *Proceedings of the International Conference on Software Engineering and Mobile Application Modelling and Development*, pages 1–5. IET, 2012.

[Krishna, 2011] Vinay Krishna. My experiments with tdd. `http://www.scrumalliance.org/community/articles/2011/may/my-experiments-with-tdd`, 2011. [Online; accessed July 2., 2014].

[Krishnan & Kellner, 1999] Mayuram S. Krishnan & Marc I Kellner. Measuring process consistency: Implications for reducing software defects. *IEEE Transactions on Software Engineering*, 25(6):800–815, 1999.

[Krishnan *et al.*, 2000] Mayuram S Krishnan, Charlie H Kriebel, Sunder Kekre, & Tridas Mukhopadhyay. An empirical analysis of productivity and quality in software products. *Management Science*, 46(6):745–759, 2000.

[Ktata & Lévesque, 2010] Oualid Ktata & 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*, pages 101–107. ACM, 2010.

[Letouzey & Ilkiewicz, 2012] Jean-Louis Letouzey & Michel Ilkiewicz. Managing technical debt with the sqale method. *IEEE Software*, 29(6):44–51, 2012.

[Machowski & Dale, 1998] Francis Machowski & Barrie G Dale. Quality costing: An examination of knowledge, attitudes, and perceptions. *Quality Management Journal*, 5(3), 1998.

[Martin, 2009] Robert C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall, 2009.

[McCall *et al.*, 1977] Jim A McCall, Paul K Richards, & Gene F Walters. *Factors in software quality*. General Electric, National Technical Information Service, 1977.

[McConnell, 2004] Steve McConnell. *Code Complete: A Practical Handbook of Software Construction*. Microsoft Press, 2004.

[McConnell, 2008] Steve McConnell. Managing technical debt. *Construx Software Builders*, 2008.

[McFeeley, 1996] Bob McFeeley. Ideal: A user's guide for software process improvement. Technical report, DTIC Document, 1996.

[McLeod & MacDonell, 2011] Laurie McLeod & Stephen G MacDonell. Factors that affect software systems development project outcomes: A survey of research. *ACM Computing Surveys*, 43(4):24, 2011.

[Mens & Tourwé, 2004] Tom Mens & Tom Tourwé. A survey of software refactoring. *IEEE Transactions on Software Engineering*, 30(2):126–139, 2004.

[Miccolis *et al.*, 2001] Jerry A Miccolis, Kevin Hively, Brian W Merkley, & Tillinghast-Towers Perrin. *Enterprise risk management: Trends and emerging practices*. Institute of Internal Auditors Research Foundation, 2001.

[Mockus & Weiss, 2001] Audris Mockus & David M Weiss. Globalization by chunking: a quantitative approach. *IEEE Software*, 18(2):30–37, 2001.

[Mockus, 2009] Audris Mockus. Succession: Measuring transfer of code and developer productivity. In *Proceedings of the 31st International Conference on Software Engineering*, pages 67–77. IEEE Computer Society, 2009.

[Mohapatra & Gupta, 2011] Sanjay Mohapatra & Divya Kumar Gupta. Finding factors impacting productivity in software development project using structured equation modelling. *International Journal of Information Processing and Management*, 2(1):90–100, 2011.

[Monden *et al.*, 2002] Akito Monden, Daikai Nakae, Toshihiro Kamiya, Shin-ichi Sato, & K-i Matsumoto. Software quality analysis by code clones in industrial legacy software. In *Proceedings of the 8th IEEE Symposium on Software Metrics*, pages 87–94. IEEE, 2002.

[Morse, 1993] Wayne J Morse. A handle on quality costs. *CMA - the Management Accounting Magazine*, 67(1):21–21, 1993.

[Moses, 2009] John Moses. Should we try to measure software quality attributes directly? *Software Quality Journal*, 17(2):203–213, 2009.

[Nielsen, 1994] Jakob Nielsen. Usability inspection methods. In *Conference Companion on Human Factors in Computing Systems*, pages 413–414. ACM, 1994.

[Ottaviani & McConnell, 2006] Stefano Ottaviani & Steve McConnell. *Software Estimation: Demystifying the Black Art*. Microsoft Press, 2006.

[Ould, 1999] Martyn A Ould. *Managing Software Quality and Business Risk*. John Wiley & Sons, 1999.

[Petrasch, 1999] Roland Petrasch. The definition of "software quality": A practical approach. In *Proceedings of the 10th International Symposium on Software Reliability Engineering*, pages 33–34, 1 1999.

[Plunkett & Dale, 1983] JJ Plunkett & BG Dale. Quality costing: a study in the pressure vessel fabrication section of the process plant industry. *Quality Assurance*, 9:93, 1983.

[Plunkett & Dale, 1987] JJ Plunkett & BG Dale. A review of the literature on quality-related costs. *International Journal of Quality & Reliability Management*, 4(1):40–52, 1987.

[Pomeroy-Huff *et al.*, 2009] Marsha Pomeroy-Huff, Robert Cannon, Timothy A Chick, Julia Mullaney, & William Nichols. The personal software processsm (pspsm) body of knowledge, version 2.0. Technical report, DTIC Document, 2009.

[Popli & Chauhan, 2014] Rashmi Popli & Naresh Chauhan. Cost and effort estimation in agile software development. In *Proceedings of the International Conference on Optimization, Reliabilty, and Information Technology*, pages 57–61. IEEE, 2014.

[Porter & Rayner, 1992] Leslie J Porter & Paul Rayner. Quality costing for total quality management. *International Journal of Production Economics*, 27(1):69–81, 1992.

[Power, 2013] Ken Power. Understanding the impact of technical debt on the capacity and velocity of teams and organizations: viewing team and organization capacity as a portfolio of real options. In *Proceedings of the 4th International Workshop on Managing Technical Debt*, pages 28–31. IEEE, 2013.

[Pyzdek & Keller, 2003] Thomas Pyzdek & Paul A Keller. *The Six Sigma Handbook*. McGraw-Hill New York, 2003.

[Ramasubbu & Balan, 2012] Narayan Ramasubbu & Rajesh Krishna Balan. Overcoming the challenges in cost estimation for distributed software projects. In *Proceedings of the 34th International Conference on Software Engineering*, pages 91–101. IEEE Press, 2012.

[Ramasubbu & Kemerer, 2013] Narayan Ramasubbu & Chris F Kemerer. Towards a model for optimizing technical debt in software products. In *Proceedings of the 4th International Workshop on Managing Technical Debt*, pages 51–54. IEEE, 2013.

[Raza & Faria, 2014] Mushtaq Raza & João Pascoal Faria. A model for analyzing estimation, productivity, and quality performance in the personal software

process. In *Proceedings of the 2014 International Conference on Software and System Process*, pages 10–19. ACM, 2014.

[Roehm *et al.*, 2012] Tobias Roehm, Rebecca Tiarks, Rainer Koschke, & Walid Maalej. How do professional developers comprehend software? In *Proceedings of the International Conference on Software Engineering*, pages 255–265. IEEE Press, 2012.

[Rombach *et al.*, 2008] Dieter Rombach, Jürgen Münch, Alexis Ocampo, Watts S Humphrey, & Dan Burton. Teaching disciplined software development. *Journal of Systems and Software*, 81(5):747–763, 2008.

[Ross, 1977] Douglas T Ross. Structured analysis (sa): A language for communicating ideas. *IEEE Transactions on Software Engineering*, (1):16–34, 1977.

[Rothman, 2006] J Rothman. An incremental technique to pay off testing technical debt. `http://www.jrothman.com/2006/01/an-incremental-technique-to-pay-off-testing-technical-debt-2/`, 2006. [Online; accessed August 5., 2014].

[Rubin, 1994] Jeffrey Rubin. *Handbook of Usability Testing*. John Wiley & Sons, 1994.

[Sandoval-Chávez & Beruvides, 1998] Diego A Sandoval-Chávez & Mario G Beruvides. Using opportunity costs to determine the cost of quality: a case study in a continuous-process industry. *The Engineering Economist*, 43(2):107–124, 1998.

[Sarigiannidis & Chatzoglou, 2013] Lazaros Sarigiannidis & Prodromos D Chatzoglou. Quality vs risk: An investigation of their relationship in software development projects. *International Journal of Project Management*, 2013.

[Savolainen *et al.*, 2012] Paula Savolainen, Jarmo J Ahonen, & Ita Richardson. Software development project success and failure from the supplier's perspective: A systematic literature review. *International Journal of Project Management*, 30(4):458–469, 2012.

[Schiffauerova & Thomson, 2006] Andrea Schiffauerova & Vince Thomson. A review of research on cost of quality models and best practices. *International Journal of Quality & Reliability Management*, 23(6):647–669, 2006.

[Schwaber, 2007] Ken Schwaber. *The Enterprise and Scrum*, volume 1. Microsoft Press, Redmond, 2007.

[Scotto *et al.*, 2004] Marco Scotto, Alberto Sillitti, Giancarlo Succi, & Tullio Vernazza. A relational approach to software metrics. In *Proceedings of the 2004 ACM Symposium on Applied Computing*, pages 1536–1540. ACM, 2004.

[Shihab *et al.*, 2012] Emad Shihab, Ahmed E Hassan, Bram Adams, & Zhen Ming Jiang. An industrial study on the risk of software changes. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, page 62. ACM, 2012.

[Shore, 2007] James Shore. *The Art of Agile Development*. O'Reilly Media, 2007.

[Singh & Kannojia, 2013] Brijendra Singh & Suresh Prasad Kannojia. A review on software quality models. In *Proceedings of the International Conference on Communication Systems and Network Technologies*, pages 801–806. IEEE, 2013.

[Singh, 2013] Gagandeep Singh. Metrics for measuring the quality of object-oriented software. *ACM SIGSOFT Software Engineering Notes*, 38(5):1–5, 2013.

[Slaughter *et al.*, 1998] Sandra A. Slaughter, Donald E. Harter, & Mayuram S. Krishnan. Evaluating the cost of software quality. *Communications of the ACM*, 41(8):67–73, 1998.

[Snipes *et al.*, 2012] Will Snipes, Brian Robinson, Yuepu Guo, & Carolyn Seaman. Defining the decision factors for managing defects: a technical debt perspective. In *Proceedings of the 3rd International Workshop on Managing Technical Debt*, pages 54–60. IEEE, 2012.

[Sommerville, 2010] Ian Sommerville. *Software Engineering, 9th edition*. Addison Wesley, 2010.

[Stavrinoudis & Xenos, 2008] Dimitris Stavrinoudis & Michalis Nik Xenos. Comparing internal and external software quality measurements. In *Proceedings of the 8th Joint Conference on Knowledge-Based Software Engineering*, pages 115–124. Citeseer, 2008.

[Sterling, 2013] Chris Sterling. *Managing Software Debt: Building for Inevitable Change*. Addison-Wesley, 2013.

[Stroggylos & Spinellis, 2007] Konstantinos Stroggylos & Diomidis Spinellis. Refactoring – does it improve software quality? In *Proceedings of the 5th International Workshop on Software Quality*, page 10. IEEE Computer Society, 2007.

[Subramanian *et al.*, 2007] Girish H Subramanian, James J Jiang, & Gary Klein. Software quality and is project performance improvements from software development process maturity and is implementation strategies. *Journal of Systems and Software*, 80(4):616–627, 2007.

[Sun, 2011] Haitao Sun. Knowledge for software quality control and measurement. In *Proceedings of the International Conference on Business Computing and Global Informatization*, pages 468–470. IEEE, 2011.

[Tatikonda & Tatikonda, 1996] Lakshmi U Tatikonda & Rao J Tatikonda. Measuring and reporting the cost of quality. *Production and Inventory Management Journal*, 37:1–7, 1996.

[Theodoropoulos *et al.*, 2011] Ted Theodoropoulos, Mark Hofberg, & Daniel Kern. Technical debt from the stakeholder perspective. In *Proceedings of the 2nd Workshop on Managing Technical Debt*, pages 43–46. ACM, 2011.

[Tomaszewski, 2006] Piotr Tomaszewski. *Software Development Productivity: Evaluation and Improvement for Large Industrial Projects*. PhD thesis, Blekinge Institute of Technology, 2006.

[Veerraju *et al.*, 2010] RPSP Veerraju, A Srinivasa Rao, & G Murali. Refactoring and its benefits. In *Proceedings of the International Conference on Modeling, Optimization and Computing*, volume 1298, pages 645–650. AIP Publishing, 2010.

[Wagner, 2013] Stefan Wagner. *Software Product Quality Control, 2013 Edition*. Springer, 2013.

[Wells, 1999] Don Wells. The rules of extreme programming. `http://www.extremeprogramming.org/rules.html`, 1999. [Online, accessed August 6., 2014].

[Williams *et al.*, 2010] Clay Williams, Patrick Wagstrom, Kate Ehrlich, Dick Gabriel, Tim Klinger, Jacquelyn Martino, & Peri Tarr. Supporting enterprise stakeholders in software projects. In *Proceedings of the 2010 ICSE Workshop on Cooperative and Human Aspects of Software Engineering*, pages 109–112. ACM, 2010.

[Xenos & Christodoulakis, 1997] M Xenos & Dimitris Christodoulakis. Measuring perceived software quality. *Information and Software Technology*, 39(6):417–424, 1997.

[Yu *et al.*, 1990] Weider D Yu, D Paul Smith, & Steel T Huang. Software productivity measurements. *AT&T Technical Journal*, 69(3):110–120, 1990.

[Zazworka *et al.*, 2011] Nico Zazworka, Michele A Shaw, Forrest Shull, & Carolyn Seaman. Investigating the impact of design debt on software quality. In *Proceedings of the 2nd Workshop on Managing Technical Debt*, pages 17–23. ACM, 2011.

[Zazworka *et al.*, 2013] Nico Zazworka, Clemente Izurieta, Sunny Wong, Yuanfang Cai, Carolyn Seaman, Forrest Shull, et al. Comparing four approaches for technical debt identification. *Software Quality Journal*, pages 1–24, 2013.

[Zhang *et al.*, 2011] Min Zhang, Tracy Hall, & Nathan Baddoo. Code bad smells: a review of current knowledge. *Journal of Software Maintenance and Evolution: Research and Practice*, 23(3):179–202, 2011.

Appendix

A Descriptions for Software Quality Characteristics

Accessibility

Degree to which a wide range of software characteristics and capabilities can be used to achieve goals.

Accountability

Ability to trace the actions of an entity uniquely to the entity.

Accuracy

Level of accuracy of computations and control.

Adaptability

Ability to adapt to different hardware, software and environments.

Analyzability

Ability to identify causes for defects, impacts of planned changes and parts to be modified when applying a change, in a software.

Appropriateness recognizability

Degree to which users can recognize if a software is suitable for their needs.

Authenticity

Ability to identify and prove a resource as the one claimed.

Availability

Degree to which a software or component accessible and operational.

Capacity

Degree to which the maximum limits of a software parameter meet requirements.

Clarity

Definitions and descriptions are explicit and have enough details. The language is understandable.

Coexistence

Degree to which a software can operate efficiently while sharing a common environment and resources with other software, without impacting any other software.

Compatibility

Software can be used with different hardware configurations and among other software without problems. For developers compatibility means the ability to change components or services with minimal effort.

Confidentiality

Ability to protect the data to be only accessed by the approved users.

Correctness, Functional correctness

Degree to which a software provides the correct results with the needed precision. Correctness is usually measured by "defects per thousand lines of code".

Documentation

There is a documentation that provides clear references and information that matches the implemented software functionality.

Economy

Ability to release the software to customer with less or equal costs than defined budget.

Efficiency

Software performs in an efficient way using as few computing resources and source code as possible.

Fault tolerance

Degree to which a software or component is operational when hardware or software faults occur.

Flexibility

Amount of effort required to modify the components of an operational software.

Functional appropriateness

Degree to which the functions facilitate the accomplishment of specified tasks and objectives.

Functional completeness

Degree to which the software functions cover the specified tasks and user objectives.

Functionality, Functional suitability

Level of satisfaction of specified software functionality expected by the user.

Generality

Extent of software component (re-)usage.

Installability

Ease of installation/uninstallation of a software to a specified environment.

Integrity

Level of protection against harmful or erroneous actions performed against functions or data.

Interoperability

Ability to function, coexist and cooperate with other systems.

Learnability

Degree to which a software can be used to learn the usage of the software. Measurement concerns effectiveness, efficiency, freedom from risk and satisfaction in a specified context of use.

Maintainability

Ability to maintain a software properly it is important that the source code is comprehensible and well-structured for flexibility for changes.

Maturity

Degree to which a software or component meets reliability requirements and expectations.

Modifiability
Ability to make modifications to software source code.

Modularity
Level of well-structured architecture concerning components, in a way that making changes to certain components require minimal changes in other components.

Non-repudiation
Ability to prove that actions or events have taken place so they cannot be repudiated afterwards.

Operability
Degree to which a software has attributes that ease the operation and control.

Performance, Performance efficiency
Ability to respond and perform actions at runtime.

Portability
Dependency of a certain platform.

Productivity
Level of efficiency in the software's development process.

Recoverability
Ability to recover the state and data from interruption or failure of a software.

Reliability
Frequency of availability and degree of specified or expected operation of the software.

Repairability
Ability to fix a malfunctioning system and continue operation.

Replaceability
Ability to be replaced by another software in specified environment.

Resilience

Tolerance for occurring errors. Similar to reliability.

Resource utilization

Degree to which the amounts and types of resources used by a software meet requirements when performing its functions.

Reusability

Extent of reusable components in a software for e.g. future use in other projects.

Robustness

Level of fluent operation in different situations e.g. error-handling, crashes and calculations.

Security

Protection level against malicious attackers when the software is in operation. Consists of software and hardware, covering the whole environment.

Testability

Ease, effectiveness and success of testing established for a software or component.

Time behaviour

Degree to which the response, processing times and throughput rates of software meet set requirements when performing its functions.

Timeliness

The ability to release the software to customer before or at the time when it is defined.

Understandability

Level of well-structured source code that can be understood.

Usability

Usability describes how straightforward and intuitive a software is to use. The functional suitability, reliability and performance are tightly related to usability.

User error protection

Degree to which a software protects users from making errors.

User interface aesthetics

Degree to which the usage of a user interface is satisfying.

Validity

Level of congruency with the specified product qualities.

Visibility, Transparency

Level of available information about the software project.