MASTER THESIS

Thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Engineering at the University of Applied Sciences Technikum Wien - Degree Program Software Engineering

A Model for Measuring Maintainability Based on Source Code Characteristics

By: David Leitner, BSc

Student Number: 1510299010

Supervisors: Dipl.-Ing. (FH) Arthur Zaczek
Paul Rohorzka

Vienna, May 4, 2017

FH University of Applied Sciences
TECHNIKUM WIEN

Declaration

“As author and creator of this work to hand, I confirm with my signature knowledge of the relevant copyright regulations governed by higher education acts (for example see §§21, 42f and 57 UrhG (Austrian copyright law) as amended as well as §14 of the Statute on Studies Act Provisions / Examination Regulations of the UAS Technikum Wien).

In particular I declare that I have made use of third-party content correctly, regardless what form it may have, and I am aware of any consequences I may face on the part of the degree program director if there should be evidence of missing autonomy and independence or evidence of any intent to fraudulently achieve a pass mark for this work (see §14 para. 1 Statute on Studies Act Provisions / Examination Regulations of the UAS Technikum Wien).

I further declare that up to this date I have not published the work to hand nor have I presented it to another examination board in the same or similar form. I affirm that the version submitted matches the version in the upload tool.“

Vienna, May 4, 2017 Signature

Kurzfassung

Da Softwaresysteme immer öfter unternehmenskritische Geschäftsfelder abdecken, gilt es das Auftreten von Fehlverhalten zu minimieren. Gleichzeitig müssen sich Unternehmen kontinuierlich neu ausrichten, um bei einem sich ständig wandelnden Markt wettbewerbsfähig zu bleiben. Dies macht Veränderungen in Softwaresystemen unvermeidlich, wodurch deren Wartbarkeit an Bedeutung gewinnt. Daraus ergibt sich das Interesse, diese Eigenschaft in bestehenden Softwaresystemen messbar zu machen. Aus diesem Grund analysiert diese Arbeit bestehende Herangehensweisen zur Messung von Wartbarkeit. Aus der Untersuchung dieser Methoden werden Anforderungen für einen modellbasierten Ansatz abgeleitet und definiert. Diese Voraussetzungen bilden die Grundlage für das vorgestellte Modell, welches auf der Korrelation zwischen der Wartbarkeit von Softwaresystemen und der technischen Qualität des Quellcodes basiert. Das entwickelte Modell wird in weiterer Folge auf bestehende Softwareprojekte angewendet, um die gewonnenen Ergebnisse zu bewerten und zu diskutieren. Das Ziel dieser Arbeit ist es, aufgrund dieser Erkenntnisse zu prognostizieren, ob und mit welcher Genauigkeit die Wartbarkeit von Softwaresystemen gemessen werden kann. Final werden die Vorteile des vorgestellten Modells im Vergleich zu bereits etablierten Ansätzen zur Messung von Wartbarkeit demonstriert.

Schlagworte: Wartbarkeitsmodell, Quellcodeeigenschaften, Softwaremetriken, Ursachenanalyse, Wartbarkeitsvisualisierung

Abstract

As software systems often cover today's companies' business-critical parts, it is a high priority to keep the number of defects close to zero. At the same time, companies must continually adapt to a changing market to remain competitive, which is why adjustments are indispensable. This circumstance points out the importance of a high level of quality to ensure maintainability and leads to a need for methodologies to investigate and rate these properties in existing software systems. This thesis analyses different kinds of existing attempts to measure maintainability and defines requirements for a model-based approach. The present model is based on the correlation between the maintainability of software systems and the technical quality of source code. Thus, the model uses selected Source Code Characteristics which have proven their importance on the quality of source code. The model introduced is applied to existing software projects to evaluate and discuss the given results. The goal of this thesis is to outline these findings and to determine whether and how precisely the given results can measure the maintainability of software systems. Finally, the advantages of the presented model are demonstrated in comparison to established approaches for measuring maintainability.

Keywords: maintainability model, source code characteristics, software metrics, root cause analysis, maintainability perspectives

Acknowledgements

I would like to express my deepest appreciation for my lecturer and supervisor Dipl.-Ing. (FH) Arthur Zaczek who guided me through my graduate education over my last years at the University of Applied Sciences Technikum Wien. Using his in-depth, extensive knowledge and experience in the field of software development, he gave me the best possible support in completing my final thesis. Creating a scientific work is always an exchange of emotions between overwhelming confidence and exaggerated dissatisfaction. In both cases, I want to thank him for providing the strength to set me straight as well as for opening new perspectives by asking the right questions.

Further, I would like to thank and acknowledge Paul Rohorzka for taking the time to support my thesis through the meaningful discussions and the valuable advices during our daily cooperation. I appreciate his technical accomplishments and convincing personality. Above all, I value his way of dealing with different opinions and admire his incredible knowledge in software design and software testing. It was a pleasure working with him.

I would also like to express my gratitude for my colleague Matthias Lischka for contributing to this thesis by providing highly valuable feedback and by detecting some missing information. Overall, I would like to thank all the developers, testers, and the other brilliant minds at TechTalk for sharing their countless experiences with me. Without their expertise, this work would have never been possible.

Last, but not least, I would like to thank my family for their constant support, for accompanying me at every step of this incredible journey, and especially, for putting their faith in me and being a source of motivation during these years. I want to extend my thanks to my friend Thomas, for our many midnight coffee chats, during which we discussed our topics and supported each other. Finally, I want to thank my girlfriend Tamara, who does not have much to do with software engineering, but understands what it means to give me a wonderful look at the world outside of it.

Contents

Introduction

> *You can't manage what you can't control, and you can't control what you don't measure. To be effective software engineers or software managers, we must be able to control software development practice. If we don't measure it, however, we will never have that control.*
>
> – Tom DeMarco, *Controlling Software Projects [35]*

The maintainability of software systems is a well-researched topic. A large number of publications report the importance of this matter and state the significance of good maintainability in software development [2, 31, 94]. Most of these investigations are based on the assumption that software systems with good maintainability characteristics will facilitate adaptions and be less susceptible to faults [86]. This leads to the conclusion that good software maintainability can result in substantial cost savings [15, 59]. Although the importance of maintainability is known, such knowledge does not have a major impact on the implementation of software systems. As recognised by Bass et al. [9], the redesign of systems is usually not triggered by inadequate functionality – the replacements are functionally identical in most cases – but due to insuperable difficulties in maintainability, portability or scalability. Similar problems caused by poor maintainability also confirm why 70 percent of the total software expenditure is being incurred to maintaining the existing systems, a fact widely accepted in the software community [15, 59, 90, 164].

The causes of these problems are manifold but can be traced back to the subjectivity of quality. In this context, the stress ratio in the triangle of time, costs and quality is the central factor for success or failure in software development. Quality, and in particular maintainability, differs distinctively from the other factors, as they cannot be changed immediately by taking management measures and because revising existing deficiencies can be difficult [148]. Additionally, there is no universal measure to rate the maintainability of a software system [86]. Consequently, the problem of maintainability in software has been articulated very appropriately by DeMarco [35] as, "you cannot control what you cannot measure". For this reason, a large number of attempts have already been undertaken to make software maintainability measurable [62, 122]. Even though these methods and tools differ significantly in their approach and implementation, they all result in a common weak point of being limited in usefulness [62].

In this thesis different definitions of maintainability are explored and a variety of established models and metrics for its measurement investigated. These analysed and discussed approaches have various drawbacks caused by wrongly assumed objectives [62]. As a result, research has long been focused on providing a single number to compare and rate software systems [91, 122].

From today's point of view, maintainability measurements should actively contribute to improvements during development, instead of merely being means for assessment. The provision of a single result without any possibility to obtain detailed insights does not support this goal. For this reason, the presented model enables root-cause analysis to improve the traceability of maintainability issues. As such, a causative linking between source code characteristics and the resulting maintainability ratings is given.

> **Hypothesis I:** Existing approaches suffer from traceability loss when computing maintainability scores of software systems. To allow root-cause analysis, the proposed model defines a bijective mapping between maintainability indicators and Source Code Characteristics.

My work is additionally motivated by the desire to provide a model that forces different stakeholders of a software system to keep track of maintainability. Existing approaches do not focus on providing different perspectives to offer a specific level of detail for maintainability characteristics of a software system. Owing to this weakness, a different attempt is necessary.

> **Hypothesis II:** Proposals to rate the maintainability of software systems provide an insufficient level of detail for a given analysis task at hand. Thus, the presented model provides useful perspectives for different stakeholders of an investigated software system.

In the first two chapters of this thesis, the foundation of the presented model is defined by providing a common methodology of software maintainability that is used during this work. In addition, existing state-of-the-art and well-established approaches for measuring software maintainability are presented and compared. A final summary describes the strengths and weaknesses of these given strategies and explains differences in the requirements and goals of the presented model. Based on the definition of the given requirements, the new maintainability model itself is drafted in chapter 3. This chapter contains the definition of the model's perspectives and the linking between them. Furthermore, a suggested visualisation and insights into the foreseen extensibility are given. As the model contains a technology-specific part, a C# implementation

is presented in chapter 4. Tools and a selection of metrics that are used to apply the model to this particular technology are introduced in this chapter as well. The final implementation is applied to real-world projects with different conditions. The gained results of the introduced model are evaluated in detail by comparing them with maintainability ratings of existing approaches. This comparison leads to a final discussion on the strengths and weaknesses of the presented maintainability model. Additionally, the efficiency and usefulness of measuring maintainability, are covered. Finally, the thesis is summed up in the conclusion section; also presented in this context is how future work can be focused on using the proposed model in the real world, by integrating it into the quality assurance pipeline of software development.

This thesis includes the following contributions to the research on measuring the maintainability of software in the field of computer science:

- an analysis of strengths and weaknesses for existing approaches to measure maintainability;
- a classification of source code characteristics, which tend to have a huge impact on the maintainability properties of a software system;
- a maintainability model which offers different perspectives to provide focused insight for different stakeholders; and
- the application and evaluation of the presented model on real-world projects and a comparison by results from established maintainability measurement approaches.

1 Maintainability of Software Systems

1.1 Maintenance and Maintainability

At the beginning of software development in the 1950s and early 1960s, maintenance represented only a small part in the software development life cycle [13]. This changed as systems began to be in use longer and maintainability thereby began to play an important role. In many sectors of the industry, the maintenance of software has become more costly than the initial development [164].

Today, controlling and managing changes is one of the greatest challenges we have to deal within software engineering [4, 59], and the importance of maintainability is widely accepted [59]. Therefore, it is essential to provide a definition of terms that are used in the context of this thesis. In many cases [12, 59, 86, 135], the definition provided by the IEEE Standard Glossary of Software Engineering Terminology [134] is used. As this definition is well established in the fields of software engineering, it is also used in this thesis. Maintainability is defined as

> [...] the ease with which a software system or component can be modified to correct faults, improve performance or other attributes, or adapt to a changed environment [134].

According to this definition, maintainability is a property of software systems while maintenance is defined as the corresponding process after delivery [134].

In the context of maintainability, the term Code Smell is often used, which describes a symptom of poor design and implementation choices [153]. The concept was mainly influenced by Fowler et al. [52], which defines it as a "[...] surface indication that usually corresponds to a deeper problem in the system" [98]. Several studies have proven that Code Smells have a negative effect on the maintainability of software systems [145, 153, 163].

1.2 Software Systems

This thesis is about the maintainability of software systems. Therefore, it is essential to provide a common understanding of what is meant by 'software systems' in the course of this thesis. Grubb and Takang [59] describe the confusion between programmes and software systems as

common misconception. Additionally, McDermid [106] confirms this assumption, as he defines software systems as an aggregation of programmes, documentation and "[...] operating procedures by which computers can be made useful to man" [106]. Even though software systems contain more than source code, the main focus of this thesis will be on this part, as an approach to measure maintainability on the basis of Source Code Characteristics is introduced. Furthermore, it is worth mentioning that this thesis makes no distinction between software and software systems.

1.3 Stakeholders

According to Rozanski and Woods [138], software development is always driven by requirements. These requirements are further defined by the needs of users. Even if the definition of *user* varies, software development is always about this described pattern. Another important insight is given by the statement that users "[...] are not limited to those who use [...]" a software system [138]. People who could have different tasks related to the software system (such as developing, testing, maintaining or using it) are called stakeholders [138]. A more general definition is given by the ISO/IEC 42010 standard, which defines stakeholders as "individual, team, organization, or classes thereof, having an interest in a system" [72].

1.4 Metrics

The term Software Metrics covers a variety of different activities in the area of software measurement [48]. Kan [77] proposes dividing it into three categories: (i) product metrics, (ii) process metrics and (iii) project metrics. Product metrics measure the properties of a software system [77]. Examples of this measurement are size, complexity or quality. Process metrics deal with the way a software is developed and maintained. Examples of this are the effectiveness of defect removal or the effort and time to implement new features. The third group, i.e. project metrics, describe the software development environment, for example, the number of developers involved, the costs incurred or the size of the organisation.

According to this definition, the measurement of maintainability on the basis of source code characteristics is a subcategory of product metrics [77]. This work is therefore primarily focused on this aspect of software measurement. A variety of metrics has already been presented to measure this aspect, which examine different characteristics of source code. Examples are volume [62, 157], complexity [103] or cohesion [27, 41, 42], to name a few [17]. Probably the most widely recognised and used metrics are those presented by McCabe and Halstead, and their variations [17]. The metrics of the presented model are described in detail in chapter 4.

2 Analysis of Existing Approaches

A series of tools and methodologies already exists to measure and assess the maintainability of software systems. The existing approaches are presented in the following three groups: (1) Software Quality Models, (2) Debt-based Approaches and (3) Metrics-driven Approaches. This grouping is not based on any existing literary foundation but was purely designed to provide a better structure for the analysis and comparison of existing approaches.

2.1 Evaluated Approaches

2.1.1 Software Quality Models

Maintainability is often only discussed as part of the overall software system quality [16, 39, 57, 104]. Thus, most of the approaches for measuring maintainability do not only cover the maintainability of software systems, but rather are part of a complete Quality Model [16, 104]. In this thesis, the analysis is mostly focused on these maintainability parts of the considered Software Quality Models.

McCall

The model by McCall et al. [104] defines the relationship between Software Quality Attributes on the basis of a hierarchy of three levels. At the top level, a system is considered by dividing it into three Product Activities [54]. They are defined as (1) Product Operations, which cover the quality of a system during its execution; (2) Product Transitions, which cover the ability of a system to handle environmental changes; and (3) Product Revisions, which are related to error correction and system adaption [46, 124]. These external Product Activities are related to internal but non-directly measurable Quality Factors. In further consequence, these factors are linked to Quality Metrics to provide measurability [46, 104].

The model considers maintainability as a Quality Factor, which is linked to four Quality Metrics: simplicity, consciousness, self-descriptiveness and modularity [46, 104]. These Quality Metrics can be measured in the form of checklists and the rating should be performed by using num-

bers from 0 (goal not reached) to 10 (excellent realisation). Still, McCall does not provide any concrete metrics or calculations for measurement [46].

Boehm

Similar to the model by McCall et al., Boehm et al. present a Quality Model as a hierarchical structure of characteristics. The quality of a system is determined by the total number of defined characteristics [16, 124].

Boehm defines the General Utility of a software system as the root of the presented model. Thus, the usefulness of a system has the highest priority and must be placed above all other characteristics. In further consequence, the model links this root node to high-level characteristics, which are defined as As-Is Utility (can be interpreted as usability), maintainability and portability [46]. The first two are linked with further sub-characteristics before they are linked to Primitive Characteristics. These nodes are determined as entry points to apply metrics [16]. However, the model does not provide any suggestions about useful and reasonable metrics [46].

The model considers maintainability as a high-level characteristic which acts as a parent of three child nodes: testability, understandability and modifiability. These child nodes are further linked to nine Primitive Characteristics: consistency, accountability, accessibility, communicativeness, self-descriptiveness, structuredness, conciseness, legibility and augmentability.

Dromey

Dromey [39] introduced a model for product quality which is based on the idea of providing a methodology which can be used for a series of different software systems [169]. In addition, Dromey makes the assumption that software does not manifest Quality Attributes [44], and thus, the model is focused on product characteristics that can be used to indicate and measure Quality Attributes [44]. As such, Dromey attempts to link Software Product Properties with Software Quality Attributes [107, 109]. The model contains maintainability as a Quality Attribute, which is linked by the Quality Carrying Properties of a product [39].

FURPS

Grady [57] originally presented the FURPS model in 1992 [107, 133]. FURPS is an acronym of the five attributes that are used to classify the quality of software systems: functionality, usability, reliability, performance and supportability [57]. These attributes contain different sub-attributes; maintainability is one and is subordinate to supportability [57]. The model was later extended

to FURPS+ by IBM [107, 133]. However, since this model does not delve into details about maintainability, it is not examined any further.

ISO/IEC 9126

The ISO 9126 family, named *Software Product Evaluation – Quality Characteristics and Guidelines for Their Use*, includes the subsection ISO/IEC 9126-1 as first part, which defines a Quality Model. This model describes six high-level Product Quality Characteristics, which are further subdivided into a total of 27 quality sub-characteristics [68]. These characteristics, which are depicted in Figure 1, are widely accepted by industry experts and academic researchers [62, 107].

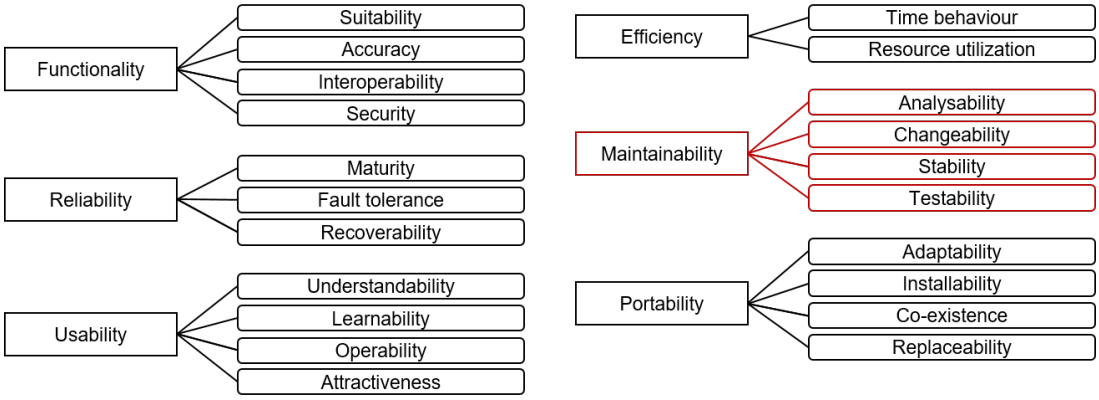

Figure 1: The ISO 9126-1 Quality Model, which is composed of six high-level quality characteristics and 21 sub-characteristics. Characteristics relevant for maintainability are marked in red.

The standard mentions maintainability as a high-level Product Quality Characteristic and complements it with four sub-characteristics [68]. ISO 9126 confirms the dependence between source code and maintainability, but lacks in providing a set of metrics based on source code properties [62, 107].

ISO/IEC 25010

The ISO/IEC 25000 family, named *Software Engineering – Software Product Quality Requirements and Evaluation* (SQUARE), is a substitution of the ISO/IEC 9126 [67]. The subsection ISO/IEC 25010 introduces a so-called Quality in Use Model [69]. In comparison to the Quality Model presented by ISO 9126, this model was mainly expanded by additional characteristics and adapted to its terminology. In detail two high-level characteristics were added: (1) compatibility, which was newly introduced to the model and (2) security, which previously was a sub-characteristic of functionality. In addition, a couple of new sub-characteristics have been added, as illustrated in Figure 2. In summary, the high-level characteristic maintainability was

restructured into five sub-characteristics (modularity, reusability, analysability, modifiability and testability).

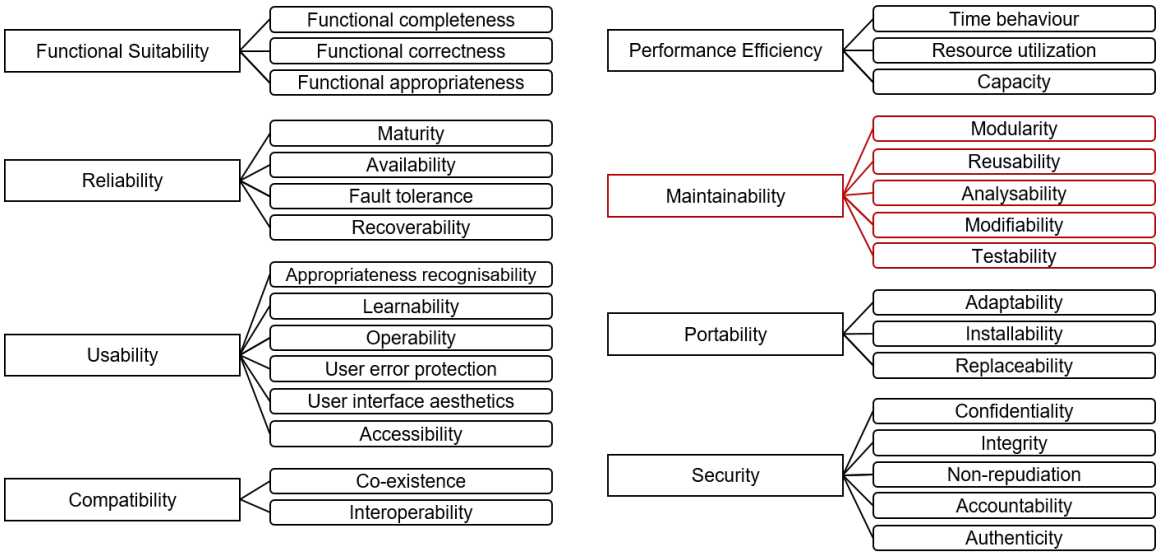

Figure 2: The ISO/IEC 25010 Quality Model is composed of eight high-level Quality Characteristics and 31 sub-characteristics. Characteristics relevant for maintainability are marked in red.

As the ISO/IEC 25010 and ISO/IEC 9126 Quality Models are strongly related, [69] provides a mapping between the sub-characteristics of both models. The mappings of the maintainability sub-characteristics are illustrated in Table 1.

ISO/IEC 25010		ISO/IEC 9126
modularity reusablity	$\Longleftrightarrow$	*no equivalent*
analysability	$\Longleftrightarrow$	analysability
modifiability	$\Longleftrightarrow$	changeability stability
testability	$\Longleftrightarrow$	testability

Table 1: Mapping between the maintainability sub-characteristics of the ISO/IEC 9126 and the ISO/IEC 25010 model [69].

2.1.2 Debt-based Approaches

In contrast to hierarchical approaches, which measure maintainability on the basis of predefined attributes, Debt-based Approaches suggest an approach which is focused on the compliance with source code requirements. Therefore, maintainability is measured based on the amount of Technical Debt in a software project [80, 107].

Technical Debt was first described by Cunningham [34] in a report of 1992. Suryanarayana et al. [149] describe it as a metaphor of a financial debt: a loan that is taken and paid back by regular instalments does not create further problems. This debt only becomes problematic when these repayments are ignored or not made, as then a penalty is imposed and the debt becomes larger and larger. This rationale also applies to software development, as it is acceptable if debts are made and *paid* back in time. But if these repayments do not occur, debts will increase and will inhibit the maintainability of the software. In the worst case, the Technical Debts reach a level at which they can no longer be repaid, which is called Technical Bankruptcy. [149]

Highsmith [64] confirms that there is a correlation between Cost of Change (CoC) and Technical Debt: the increase of Technical Debt causes an increase in CoC. Other researchers [1, 119] have reinforced the strong correlation between Technical Debt and maintainability.

SQALE

The Software Quality Assessment based on Lifecycle Expectations (SQALE) Model, presented by Letouzey and Coq [91] is organised in three hierarchical levels and formalises the relation between non-functional requirements and the quality of source code. Therefore, it maps characteristics to sub-characteristics, which in turn are linked to source code requirements [91]. The characteristics are defined as: reusability, portability, maintainability, security, usability, efficiency, changeability, reliability and testability. They are based on the ISO/IEC 9126 Quality Model, but are restructured and contain different sub-characteristics [107].

The model uses control points to measure the compliance with source code requirements and to reveal coding rule violations. Each rule defines a Remediation Index, which represents the work effort needed to correct the non-compliance [107]. SQALE comes with a set of more than 30 different control points [91]. A general overview of the final rating for the whole system or different components can be calculated based on the Remediation Indices. This rating is based on the five different levels A, B, C, D and E (A being excellent and E being poor) [91, 107].

The SQALE Model is implemented by a variety of tools. In the course of their evaluation, each demonstrates that they focus on different details and are distinguished from the initial

definition by Letouzey and Coq [91]. These diverging implementations have influenced how the requirements of the presented model have been defined. Some of the most influential SQALE implementations are explained below:

- **SonarQube**[1] is an open-source platform specialising in the analysis of code quality on a continuous basis [80]. It provides different top-level indicators for the quality of a system, such as reliability, security, maintainability, coverage and duplication. The measurement of maintainability is calculated by an implementation of the SQALE methodology [107] and visualised as a maintainability rating. This rating is based on the Technical Debt Ratio, which is according to the SonarQube documentation [147] defined as

$$TechnicalDebtRatio = \frac{RemediationCost}{CostsToDevelopOneLineOfCode * LinesOfCode}$$

  where the $RemediationCost$ is calculated by summing the *Time-to-Fix* of all violated Code Smells. SonarQube provides rules in different sets for several programming languages to gather these Code Smells. The $CostsToDevelopOneLineOfCode$ can be configured, while the $LinesOfCode$ are deduced from the codebase. Finally, the SonarQube Maintainability Rating can be calculated from the Technical Debt Ratio as described in Table 2.

Technical Debt Ratio	$\leq 5\%$	$\leq 10\%$	$\leq 20\%$	$\leq 50\%$	$> 50\%$
Maintainability Rating	A	B	C	D	E

  Table 2: Boundaries used by SonarQube to derive the Maintainability Rating of a codebase, on the basis of the Technical Debt Ratio.

- **NDepend** is a commercial tool to analyse C# and Visual Basic solutions. The tool provides a bulk of functionality, including an implementation of the SQALE methodology. The rating of a solution from A to E (with A being best and E being worst) is based on the non-compliance of predefined rules. NDepend comes with a set of default rules that can be used out of the box, but also provides the possibility to implement custom rules through the usage of code snippets written in Code Query for Language Integrated Query (CQLinq) [141]. The debt for each of these rules can be configured independently. Similar to SonarQube, the final codebase rating is calculated by summing up all these debts and putting them in relation to the estimated redevelopment effort. According to the NDepend documentation [32], the mapping between this ratio and the final rating is similar to the mapping defined in Table 2.

  Similar versions of NDepend are also available for other programming languages, such as JArchitect[2] (for Java) and CppDepend[3] (for C++).

[1] available via sonarqube.org
[2] available via jarchitect.com
[3] available via cppdepend.com

- **SQuORE** by SQUORING[4] is a commercial implementation of the SQALE methodology. It is focused on providing visualisations by a web-based dashboard, as illustrated in Figure 3. The rating from A to G (with A being best and G worst) can be observed on different levels of the codebase, such as the whole system or a single function. This supports and allows good root-cause analysis of maintainability issues.

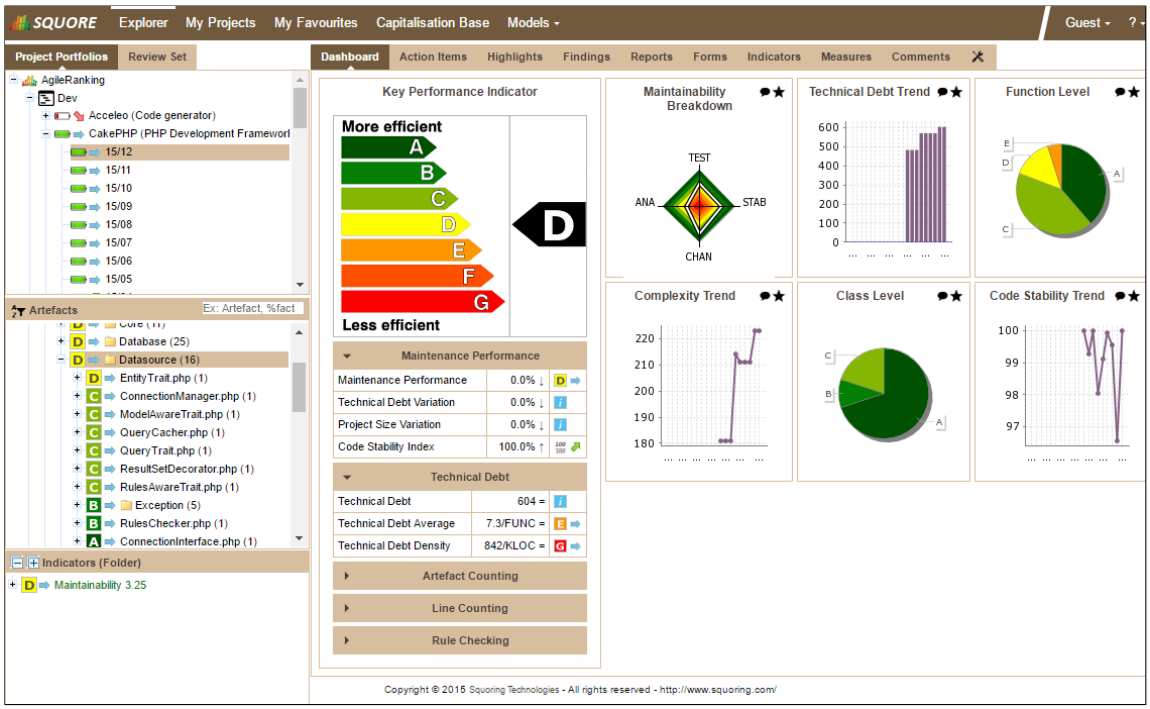

Figure 3: The SQuORE dashboard [151] shows different projects and the structure of a selected project on the left side. The Performance Indicator, Line Checks, Rules Checks and other details about the quality of the observed source code are presented on the right side.

2.1.3 Metric-based Approaches

Similar to Debt-based Approaches, but in contrast to Software Quality Models, Metric-based approaches result in concrete indicators. Thus, information about the maintainability of a system can be gathered immediately. This result is provided by some kind of calculation or metric and is defined as a number [31] or value range [62], which ensures easy comparability.

Maintainability Index

Oman and Hagemeister [122] introduced a composite metric for quantifying software maintainability [31, 160]. This Maintainability Index (MI) has been adapted multiple times and was

[4] available via squoring.com

successfully applied to a tremendous number of industrial software systems [130, 160]. It is defined as a composed number of four different source code-based metrics and is calculated by

$$MI = 171 - 5.2 * \ln(HV) - 0.23 * CC - 16.2 * \ln(aveLOC) + 50 * \sin(\sqrt{2.46 * COM})$$

where HV represents the Halstead Volume [60], CC represents the Cyclomatic Complexity [103], $aveLOC$ is defined as the average lines of code per module and COM is defined as the percentage of comments within a module [62, 122]. This fitting function was calculated by observing a large number of software systems [62, 122]. The higher the resulting MI, the better the system's maintainability.

The usage of comments in the MI is frequent point of criticism [62, 96], both because the calculation is not comprehensible [96] and because comments are denied as an indicator of good software maintainability [62]. As a result, the MI is implemented differently in various tools.

- **Visual Studio Enterprise 2017 (VS2017)**[5] provides a functionality to calculate the MI. According to its documentation [108] an adapted version without comments and a normalised result is implemented. It returns a value between 0 and 100, where a value < 10 indicates low maintainability, a value < 20 indicates moderate maintainability and a value ≥ 20 indicates good maintainability.
- **JHawk**[6] is a Java Metrics Tool that provides two different implementations of the MI. These are documented [156] as (1) MINC, which ignores the comments part and (2) MI, which is calculated as proposed by Oman and Hagemeister [31].

Probabilistic Software Quality Model

Bakota et al. [7] presented the Probabilistic Software Quality Model, which is based on the ISO/IEC 9126 maintainability attributes. It was recognised that the terminology of ISO/IEC 9126 is well established, but the standard lacks in the provision of measurements and metrics [7]. Thus, the given attributes have been extended and structured as nodes in a directed acyclic graph, as illustrated in Figure 4 [7, 107]. The measurement of these nodes is done by so-called Sensor Nodes, which measure the maintainability based on source code properties [107]. In the following, the quality of an attribute represented by a node is measured by a Goodness Function [107]. This goodness is defined as a scalar between 0 and 1, with 0 being worst and 1 being best [7].

[5] available via visualstudio.com
[6] available via virtualmachinery.com/jhawkprod.htm

The aggregation of these nodes is done by the usage of weights that were proposed by industry and academic experts in an online survey. The incoming nodes for each aggregation node are weighted by scalars that add up to 1 [7, 107].

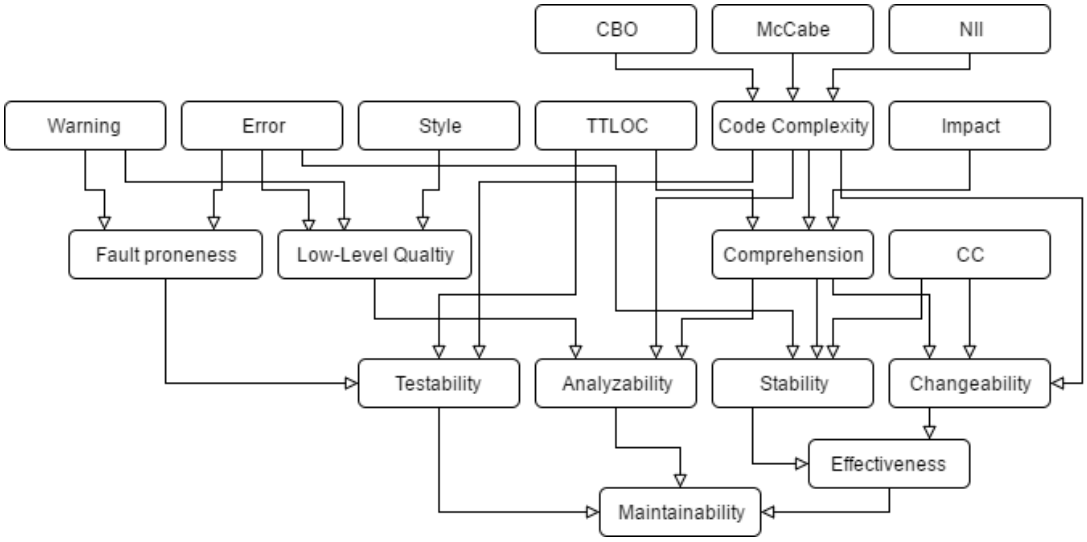

Figure 4: The Probabilistic Software Quality Model presented by Bakota et al. [7] which contains an acyclic graph based on the ISO/IEC 9126 maintainability characteristics.

SIG Maintainability Model

Heitlager et al. [62] introduced a maintainability model as an alternative to the MI presented by Oman and Hagemeister [122]. The model is based on the maintainability characteristics of ISO/IEC 9126 and the assumption that these System Quality Characteristics can be caused by Source Code Properties. This mapping is illustrated in Table 3. Source Code Properties are measured by the introduction of source code metrics. These measurements and metrics were proposed by Heitlager et al. [62] with a focus on technology-independence and simplicity. The measurement results are aggregated and mapped back to System Quality Characteristics, which are finally rated in five steps from $--$ (worst) to $++$ (best). In addition to the proposed metrics and the reuse of the ISO/IEC 9126 terminology, an improved root-cause analysis is highlighted as a strength of the model [62].

On the basis of the given model, the Software Improvement Group (SIG) offers certifications of software products as a commercial service. As such, an implementation of the model is not available for public usage [107]. However, the model is implemented as a SonarQube plugin [143] and can be downloaded for free [107].

	volume	complexity per unit	duplication	unit size	unit testing
analysability	X		X	X	X
changeability		X	X		
stability					X
testability		X		X	X

Table 3: The mapping defined by the SIG Maintainability Model between System Quality Characteristics and Source Code Properties [62].

2.2 Analysis and Comparison

The comparison of the presented approaches is done from two different perspectives. First, it is based on the three presented groupings. After that, the different characteristics of maintainability models are examined.

2.2.1 Based on Groupings

Software Quality Models

Software Quality Models provide a definition of evaluation criteria for the quality and maintainability of software. Even though these often differ, there are similarities, as shown in Table 4 [107, 133]. Both the criteria for general software quality and the sub-characteristics for maintainability are compared. The comparison shows that all models list maintainability as an important component of software quality. The criteria defined by these models have at least two advantages: (1) they help to consider quality and maintainability of software from different perspectives and (2) they are declared in a very general way, which makes them completely independent from technology and suitable for a broad range of different software projects. Therefore, these criteria are commonly used as foundation for more practical approaches [7, 62, 91]. The downside of these general approaches is that most of them cannot be directly used in practice since they do not include concrete descriptions or metrics for measurement. For the purpose of the model presented in this thesis, Software Quality Models are therefore mainly used to provide a well-defined and established terminology.

Debt-Based Approaches

Debt-based Approaches such as SQALE, assume that maintainability issues can be directly mapped to Technical Debts. The sum of these Technical Debts for a given system is the starting point for further examination, as they are set in relation to the effort that was needed to implement the whole system. This ratio is used as a final rating of maintainability. At best, there is no Technical Debt and, in the worst case, Technical Debts are as high as the effort needed to rewrite the entire system. The calculation is therefore relatively simple and maintainability issues can be detected immediately. An extension of these approaches can be easily achieved by adding further rules to verify maintainability aspects. Even if a grouping of the criteria can be made, a generalisation of maintainability aspects is only provided to a limited extent. Thus, the final rating is directly mapped to the given rules without the possibility of intermediate steps.

Metric-Based Approaches

Metric-based Approaches differ strongly from each other in their structure and results. While the SIG Maintainability Model and the Probabilistic Software Quality Model aggregate multiple metrics hierarchically, the MI consists of a single equation. The latter approach has the advantages that it can be calculated in a straightforward manner for multiple projects and that the computed results can be compared easily. Thus, this approach provides good results for comparing projects on a very abstract level. The disadvantage is that no logical conclusions can be drawn from the given results. Heitlager et al. [62] also argue that the composition of the equation is unclear. They come up with the example of Cyclomatic Complexity (CC), which is multiplied with 0.23, and state that there is no logical argument for why this figure is used [62]. Thus, they stated, that the justification of the formula among different stakeholders is a source of frustration [62]. In addition they criticise the used metrics, as comments are a bad indicator for the maintainability of source code, the Halstead Volume metric is difficult to define and not widely accepted within the software engineering community, and the Average Complexity is described as a “fundamentally flawed number” for systems that are implemented by the usage of object-oriented technology [62]. Thus, the SIG Maintainability Model and the Probabilistic Software Quality Model try to improve these weaknesses; both are focused on the linking of software maintainability aspects with metrics. Even so, the used metrics are strongly anchored in the model and are therefore difficult to adapt.

2.2.2 Based on Characteristics

Root-Cause Analysis

Root-cause analysis is a characteristic that easily allows the detection and visualisation of the reasons for maintainability violations. Therefore, some models do not only provide an overall rating for the maintainability of a system, but also allow the identification of causes for this rating. This possibility of identification allows for tackling possible issues and resolving them. Heitlager et al. discovered that the acceptance of a numerical metric increases greatly when practitioners can comprehend how the maintainability rating of a system is composed [62]. Hierarchical models such as the SIG Maintainability Model or the Probabilistic Software Quality Model have their strengths in this area. The absolute opposite is provided by the MI, since it is based on a fitting function and thus the calculated result does not allow for deriving causes from the given result [62]. System Quality Models only allow conditional consideration of their possibility for root-cause analysis, as they do not provide enough details about the practical measurement based on source code properties. Debt-based Approaches allow the linking between weighted roles and the final maintainability rating. However, due to the lack of intermediate steps and generalisation, these approaches only have weak possibilities for allowing the visualisation of root-causes for further examination.

Technology-Independence

Technology-independence is a characteristic that allows the model to be applied on different technologies without any need for adaption. This introduces the advantage that multiple projects, or a single project that contains multiple technologies, can be investigated with the same model. The MI is a good example of technology-independence, as it can be applied to multiple technologies while returning a comparable result. Another example is the SIG Maintainability Model, which was designed to be technology-independent [62] and can only use technology-independent metrics. Therefore, code duplication is simply detected by finding equal blocks over six lines (ignoring leading spaces) [62]. Although this measurement is widely applicable, its quality is highly limited. For this reason, technology-independence can also be considered as a disadvantage since it does not allow the usage of fitted measurements for a specific technology. A technology-specific approach is presented by the SQALE implementation SonarQube. This implementation provides a set of rules for different technologies and the number of rules in these sets can differ between different technologies. For example, SonarQube contains more rules for Java than for C#. Thus, more Technical Debts can be detected in Java projects and they are more likely to receive a worse maintainability rating in comparison to C# projects. Therefore, the presence of comparability and technology-specific measurements can be considered as trade-off.

Adaption

Adaption is the characteristic of a model that allows for the adjustment and extensibility of a specific task. SQALE-based implementations such as SonarQube, SQuORE or NDepend are good examples, as they provide possibilities to ignore suggested rules or to extend the maintainability evaluation with custom rules. These functionalities allow for the consideration of domain- and technology-specific peculiarities for the maintainability analysis. Others models, such as the MI or the SIG Maintainability Model have not been designed to enable adjustments, as they do not provide any information or explicitly defined entry points for possible adaptions.

Terminology

The terminology of the presented approaches is mostly defined by Software Quality Models. In particular, the nomenclature defined by the ISO/IEC 9126 standard is frequently reused [7, 62]. Heitlager et al. realised that the usage of a well-defined and established terminology facilitates communication between various stakeholders of the system [62]. Therefore, it is absolutely acceptable to reuse the terminology of existing models.

	McCall	Boehm	Dromey	FURPS	ISO/IEC 9126	ISO 25010
Quality Evaluation Criteria						
Compatibility						X
Correctness	X					
Efficiency	X	X	X		X	(X)[b]
Flexibility	X					
Functionality			X	X	X	X
Integrity	X					
Interoperability	X					
Maintainability	X	X	X	(X)[a]	X	X
Modifiability		X				
Performance				X		X
Portability	X	X	X		X	X
Reliability	X	X	X	X	X	X
Reusability	X		X			
Security						X
Supportability				X		
Testability	X	X				
Understandability		X				
Usability	X	X	X	X	X	X
Maintainability Evaluation Criteria						
Analysability					X	X
Testability					X	X
Modularity						X
Reusability						X
Modifiability						X
Simplicity	X					
Conciseness	X	X				
Self-Descriptiveness	X	X				
Modularity	X					
Consistency		X				
Accountability		X				
Accessibility		X				
Communicativeness		X				
Structuredness		X				
Legibility		X				
Augmentability		X				
Changeability					X	
Stability					X	

Table 4: Comparison of quality and maintainability evaluation criteria of the analysed Software Quality Models [107, 133]. When a particular criterion is used by a particular model, an “X” is drawn in the corresponding cell.

[a] Is defined as a sub-characteristic of supportability.

[b] ISO 25010 contains a criterion named performance efficiency.

3 A Model for Measuring Maintainability

3.1 Requirements

Guided by the investigation of the existing approaches (see chapter 2) and their shortcomings, the requirements are defined to be met by the presented model. The formulation of these requirements is inspired by the Agile Manifesto [53], as it follows the same pattern. It starts with a pursued goal and ends with a problem that is avoided. The three requirements are defined as:

The presented model provides:

1. a technology-agnostic frame which is extendible by specific dependent metrics, *over being completely restricted to any technology or being entirely abstract*.
2. an outcome that provides multiple liked perspectives to enable simple root-cause analysis of maintainability issues, *over a slightly comparable but difficult-to-interpret number*.
3. a terminology that consists of standardised terms and quality criteria which provide a common foundation for discussions among different stakeholders, *over introducing its own new terminology*.

Requirement (1) defines consent between the existence of two different types of maintainability models: technology-dependent or technology-independent. The first approach has the disadvantages that the model has to be redefined for each technology. From this it follows that projects which consist of multiple technologies can only be measured by using different models. The advantage is that used measurement methods and metrics can be high technology-specific and therefore very meaningful. In contrast, the second approach has the advantage that it can be used on different technologies, but can only use generalised and technology-independent measurements. Even so, it is nearly impossible to fully list all technologies used in today's software systems, which is why it is frivolous to assert that a metric is 100 percent technology-agnostic. On the basis of these considerations, the

decision to strike a judicious balance was made: a technology-independent part to provide a common basis through terminology and a predefined structure – which is extended by a technology-specific and fitted measurement.

Requirement (2) defines that the presented model is not intended to compare systems, but to obtain different perspectives of the maintainability weaknesses of the considered codebase. This requirement results from the assumption, that it is not informative to express the maintainability of a system in a single number. According to Kuipers and Visser [83], this results in the following problems: (i) the root-cause for a bad result is difficult to understand and (ii) the acceptance of a measurement result decreases, if it does not provide enough details to determine the causes.

Another advantage of an outcome which provides different perspectives to allow root-cause analysis is that different abstractions can be used for different stakeholders. For example, developers are more likely to be interested in technical details of maintainability issues, while customers only need a general overview of their software system's maintainability in most cases. Thus, it is an important requirement that the presented model supports this variety of different perspectives.

Requirement (3) ensures that the presented model uses a terminology that is standardised and widely accepted. An established terminology facilitates communication concerning maintainability between different stakeholders, as noticed by Heitlager et al. [62]. Therefore, there is no need to introduce a different nomenclature to well-understood terms.

The numbering of the requirements should not be considered as a ranking of their importance. There is no prioritisation defined within the given requirements and the stated numbers are only used for referencing.

3.2 Structure

The model provides four different perspectives for the consideration of maintainability. These perspectives represent differently generalised properties of maintainability and are linked to each other.

3.2.1 Perspectives

System Quality Characteristics

System Quality Characteristics provide the most abstract perspective on maintainability properties in the presented model. They should not represent technical details but have to be defined

in a well established terminology, which is well known in the technical context and can be interpreted by different stakeholders. The term System Quality Characteristics is based on the Product Quality Characteristics defined by the ISO/IEC 25010 Quality Model (see section 2.1.1) [69]. This standardised model contains eight Product Quality Characteristics which are further divided into sub-characteristics [69]. One of these Product Quality Characteristics is maintainability.

The ISO/IEC 25010 characteristics and sub-characteristics are widely used and accepted in the scientific context [92, 93, 120]. Additionally, they can be considered technology-independent since they are intended to be applied to computer systems and software products [69]. Therefore, the sub-characteristics of maintainability are used as the System Quality Characteristics of the presented model. These abstract properties define a solid foundation to discuss and outline the maintainability of a measured system and are defined as:

- **Modularity** defines the “degree to which a system or computer programme is composed of discrete components such that a change to one component has minimal impact on other components” [69].
- **Reusability**, defined by the “degree to which an asset can be used in more than one system, or in building other assets” [69].
- **Analysability** defines the “degree of effectiveness and efficiency with which it is possible to assess the impact on a product or system of an intended change to one or more of its parts, or to diagnose a product for deficiencies or causes of failures, or to identify parts to be modified” [69].
- **Modifiability** defines the “degree to which a product or system can be effectively and efficiently modified without introducing defects or degrading existing product quality” [69].
- **Testability** defines the “degree of effectiveness and efficiency with which test criteria can be established for a system, product or component and tests can be performed to determine whether those criteria have been met” [69].

Source Code Characteristics

The perspective of Source Code Characteristics is defined by technical terms and describes quality properties of the source code. These technical characteristics give a first insight into hotspots of the source code but are generically defined as technology-independent characteristics. There is a large body of subject literature about the properties of maintainable source code, but there is, unfortunately, no generally accepted definition of them [52, 101, 105, 157]. Based on similarities, guidelines, and best practices described in the literature they have been grouped into hypernyms. This grouping process has resulted in a list of the following six Source Code Characteristics:

- **Cohesion**, defined by the degree of the common bond of elements that are combined into one unity [10, 167]. In this case, the term element may be a source code element at different levels, such as a function, procedure or module [37]. As a metaphor, cohesion is often referred to as the glue which holds the source code elements together, and which defines the strength of their common bond [14, 37, 126, 127].
- **Coupling** is defined as the degree of interdependence between the source code elements [70], or the strength of relationship between them [71]. The relevance of this Source Code Characteristic arises from the fact that changes made to a specific part of the software should affect the rest of the application as little as possible [37]. Code with low coupling is therefore easier to maintain.
- **Complexity** is a highly controversial quality feature. While one developer could classify a source code as highly complex, another could judge it as straightforward [157]. Nevertheless, the importance of avoiding complexity can be found in almost all guidelines of maintainable code [3, 19, 101, 105, 157] and there are scientific approaches of linking software complexity to its Maintainability [90]. The complexity of the source code is, in most cases, associated with its structuring or sequence logic [157].
- **Size** is an important property of maintainability for source code. A correlation between the size of a codebase and its maintainability is described by Visser et al. [157], among others. As size is also an important factor for maintainability on different layers, the way that the balance between components has an impact on a system's maintainability is also investigated [157]. Additionally, small units and modules simplify the understanding and maintainability of the source code [101, 105]. Because of these different effects, the general term *size* was chosen to name this Source Code Characteristic.
- **Duplication** makes software maintenance more difficult since changes must be implemented in several places [76]. There are two basic types of duplication: source code parts which are textually identical (further divided into Type 1 to 3) or functionally identical (defined as Type 4) [137]. Both are declared by the Source Code Characteristic duplication. Fowler et al. [52] describe duplication as Code Smell and as an indicator for poor maintainability.
- **Coverage** is another indicator of the maintainability of source code, as it can make development predictable and less risky [157]. Other models, such as [62], also use it as an indicator of good maintainability. Thus, this Source Code Characteristic includes the coverage of automated tests on different layers (such as unit, integration or User Interface (UI) tests [154]).

Source Code Partitions

The characteristics of the source code are not always mapped to the complete codebase of a software system. They are mapped to different divisions, called Source Code Partitions. For example, in Java it is possible to examine classes or methods to observe characteristics of maintainability. For this reason, there are obviously at least two different types of Source Code Partitions in this example: classes and packages. Although the above example suggests that this partitioning is technology-dependent, similar concepts are found in different technologies which differ only by notation [157]. Therefore, the used terminology is independent of the given technology and language type and is collected from the literature [20, 150, 157, 161]. Thus, this perspective is expressed in the following Source Code Partitions:

- A **unit** is the smallest possible group of functionality that can be executed independently [157]. Thus, it is not possible to invoke only parts of a unit, since its execution is only possible as a whole [157].
- A **module** provides the smallest possible aggregation of units [157]. It allows the grouping of functionality or data and can provide a possibility to hide selected elements from the outside [161].
- A **component** is the Source Code Partition of a system based on a given organisation structure or software architecture [20, 157]. It has properties such as contractually specified interfaces, explicit context dependencies and the possibility of an independent deployment [150]. Chonoles and Schardt [30] describe a component as a subsystem with internal classes which provide a public interface that describes which operations can be invoked – but, not how they are implemented.
- A **codebase** is a collection of source code, which is usually stored in a single repository, can be compiled and deployed independently, and is maintained by one team [157]. A complete system can therefore consist of multiple codebases.

Even if the above-mentioned Source Code Partitions are designed to be technology-independent, it should be emphasised that the terminology can be extended or abridged, if the used technology can only be mapped with great effort or drawbacks. This is not a strict guideline, but should be considered a well-established suggestion.

Sensor Ports

Sensor Ports are the most specific perspective of the presented model, as they provide distinct entry points for technology-specific measurements. The naming was inspired by Sensor Nodes presented by Bakota et al. [7]. As Sensor Ports serve as placeholders in the introduced model, it is only determined where these entry points are located in the model, but not how they are

implemented. Thus, the selection of suppliers for the measurement of source code depends on the technology-specific implementation of the model and can be chosen individually. They represent the elements of the model with the best possibility for obtaining the most detailed information about maintainability issues, as they are placed on the lowest possible level of abstraction. Therefore, they should return specific measurement information and descriptions, in addition to a distilled rating. This additional information helps to assess potential issues.

3.2.2 Linking

The introduced perspectives are linked to each other based on the given requirements. This linking is illustrated in Figure 5 and allows for causes to be traced in both directions – from the System Quality Characteristics to Sensor Ports, and vice versa. This condition allows excellent root-cause analysis and traceability. This chapter describes the linking of the different perspectives of the presented model.

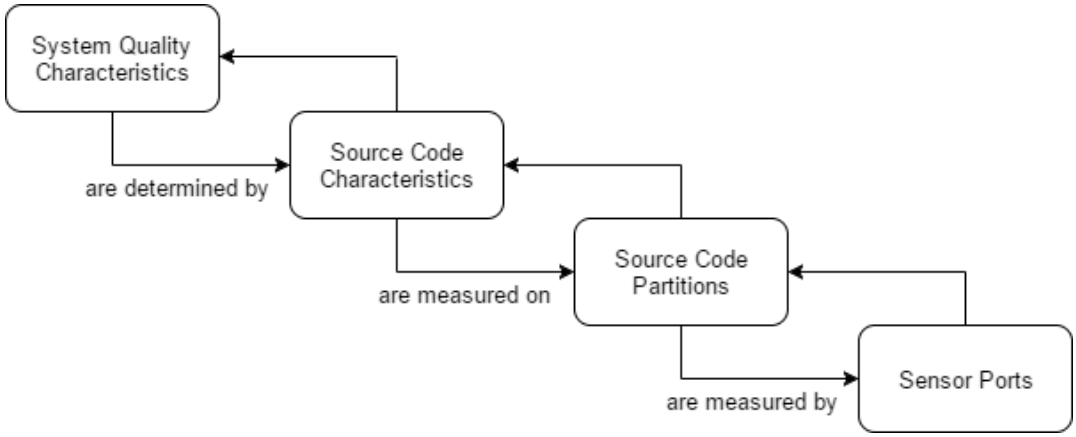

Figure 5: The presented model maps abstract System Quality Characteristics, which are defined by the terminology of ISO/IEC 25010 Quality Model, into technology specific Sensor Ports. In the first step, System Quality Characteristics are mapped to Source Code Characteristics. In the second step, these Source Code Characteristics can be measured on Source Code Partitions by the application of Sensor Ports.

Linking between System Quality Characteristics and Source Code Characteristics

The foundation of the model is given by linking System Quality Characteristics and Source Code Characteristics. These two perspectives are mapped to each other by a matrix, as illustrated in Figure 6. This simple structure is inspired by Heitlager et al. [62] and their presented SIG Maintainability Model. Even if the structure is similar, the elements differ greatly from each other.

This simple foundational mapping already supports the given requirements, as it provides different perspectives of the model for various stakeholders and supports basic root-cause analysis.

The mappings between both characteristics are illustrated in Figure 6 and discussed individually for each Source Code Characteristic:

- **Cohesion** was measured by Rosenberg [136] by using Lack of Cohesion of Methods (LCOM) and analysed to have high impact on maintainability, understandability and reuse. Thus, the Source Code Characteristic cohesion is mapped to the System Quality Characteristics analysability and reusability.
- **Coupling** was also measured by Rosenberg [136]. By using the Coupling between Object Classes (CBO) metric the impact of coupling on a codebase was declared as testing efforts, understandability, maintainability and reuse. Even if the final conclusion states that there was an impact on reuse, it was emphasised in other parts of the paper, that coupling has a big impact on modularity. This finding was confirmed by citing Chidamber and Kemerer [27], Hudli et al. [66], Lee et al. [89] and others. Therefore, it can be assumed that reuse was used as a generalised term for modularity. Additionally, according to Poulin [131], coupling (originally as dependencies) also plays an important role in the reusability of components. Therefore, the model is also supplemented by this mapping. The presented model therefore maps the Source Code Characteristic coupling to the System Quality Characteristics modularity, analysability, reusability and testability.
- **Complexity** is part of the maintainability model presented by Heitlager et al. [62]. It is declared as complexity per unit and linked to the ISO/IEC 9126 criteria (see section 2.1.1), changeability and testability. While testability is still part of the ISO/IEC 25010 Quality Model, changeability was reworded as modifiability [69]. Therefore, the Source Code Characteristic complexity is mapped to the System Quality Characteristics changeability and modifiability.
- **Size** is mentioned twice in the maintainability model presented by Heitlager et al. [62]. It is declared as volume and unit size and linked to analysability and testability. These mappings are reused in the presented model. Even if a small module size is often defined as an indicator for good reusability [132, 139], researchers such as Caldiera and Basili [21] disagree and argue that a module must be big enough to justify the costs of integration [131]. Due to these contradictory assumptions about whether and how size impacts reusability, the model avoids a mapping to this criterion.
- **Duplication** is again linked by Heitlager et al. [62] the to ISO/IEC 9126 criteria analysability and changeability. According to the more recent ISO 25010, changeability can be translated to modifiability. Additionally, modularity is defined by ISO 25010 as the property of a codebase that allows changes to one component which has a minimal impact on other components. It can be derived from this definition that modularity is closely linked

to duplication and therefore a mapping in the presented model is justified. The Source Code Characteristic duplication is finally mapped to the System Quality Characteristics analysability, modifiability and modularity.

- **Test Coverage** has a positive impact on the reusability of a component, according to Poulin [131]. Heitlager et al. [62] amend that Test Coverage also impacts analysability, modifiability and testability. Therefore, the Source Code Characteristic test coverage is linked to these four System Quality Characteristics.

	Cohesion	Coupling	Complexity	Size	Duplication	Test Coverage
Modularity		x			x	
Reusability	x	x				x
Analysability	x	x		x	x	x
Modifiability			x		x	x
Testability		x	x	x		x

Figure 6: Mapping between System Quality Characteristics and Source Code Characteristics. System Quality Characteristics are represented as rows and Source Code Characteristics are represented as columns. The mappings between those two perspectives are stated as crosses in the corresponding cell.

Linking between Source Code Characteristics and Source Code Partitions

The perspective of Source Code Characteristics provides abstract properties about the maintainability of the measured codebase. Nevertheless, they are too general to directly perform measurements on them. Thus, Source Code Partitions have been introduced, which define different divisions of a codebase that can act as entry points for further observations. Each Source Code Characteristic is linked to Source Code Partitions. These Partitions are defined once for the whole model (see section 3.2.1) and do not differ between the different Source Code Characteristics. Thus, each Source Code Characteristic is mapped to the same list of Source Code Partitions.

Linking between Source Code Partitions and Sensor Ports

Each Source Code Partition can have a list of Sensor Ports. As described in section 3.2.1, these Sensor Ports are the entry points for technology-specific measurements. In contrast to Source Code Partitions, which are defined once for all Source Code Characteristics, Sensor Ports can be chosen independently. Sensor Ports are assigned to Source Code Characteristics based on the type of measurement they perform. Furthermore, they are assigned to Source Code Partitions based on the location where they perform this measurement. Therefore, it is legitimate that Source Code Partitions can contain 0 to n Sensor Ports. As the selection of Sensor Ports depends on the project and technology, their assignment is not specified by the presented model and has to be defined under consideration of the technology from a measured project.

3.3 Rating

The foundation of all ratings in the presented model is provided by Sensor Ports. The model dictates that each Sensor Port returns a rating between 0 and 1, where a result of 0 defines that the observation of a Sensor Port yields good conditions in the source code, in contrast to 1, which suggests poor maintainability[1]. The approach to calculate this rating is dependent on the respective Sensor Ports and is therefore not defined by the presented model.

In another step, a rating for each Source Code Partition from all Source Code Characteristics is aggregated. If a Source Code Partition only contains a single Sensor Port, its rating can be reused without any adaptions. In the case of multiple Sensor Ports, their ratings are aggregated to a single value between 0 and 1. The implementation of this composition can be arbitrarily adapted and is not predetermined by the model. For example, each Sensor Port can be weighted individually. However, if no other information is given, the average is calculated.

In the next step, each Source Code Characteristic aggregates the ratings of their Source Code Partitions. This is usually done by using the average, but can also be adapted as required. Thus, each Source Code Characteristic is rated by a value between 0 and 1.

Finally, the average of all Source Code Characteristics that have an effect on a System Quality Characteristic, is calculated and mapped to the values $++$, $+$, o, $-$ and $--$. The selection of these values was inspired by Fielding [50] and is intended to clarify that these final ratings should not be aggregated further. They can be interpreted as *very good*, *good*, *moderate*, *poor*

[1] The decision that 0 is defined as good and 1 as bad indicator is the result of a long discussion, which was initiated by the fact that the Source Code Characteristics are partly contradictory to each other. Thus, little duplication (interpreted as 0) is good, but little cohesion (also interpreted as 0) is critical. Finally, 0 was chosen to indicate goodness, as programming languages such as C++ [95] use it as positive return value, which denotes success.

and *very poor*. The mapping between the calculated average, the output and its interpretation is summarised in Table 5.

Rating	Output	Interpretation
$a \leq 0.2$	$++$	very good
$0.2 < a \leq 0.4$	$+$	good
$0.4 < a \leq 0.6$	o	moderate
$0.6 < a \leq 0.8$	$-$	poor
$> a > 0.8$	$--$	very poor

Table 5: Mapping of System Quality Characteristic ratings to final outputs and their interpretations, where a is defined as the average of all affecting Source Code Characteristics.

3.4 Extensibility

The presented model was designed to be extensible for specific conditions. Thus, this section defines how perspectives allow adaptions. In general, the more abstract the perspective, the fewer changes should be made. Thus, System Quality Characteristics must stay unchanged as they have been defined by the ISO/IEC 25010 and are widely established in the field of Information and Computer Science. Also, Source Code Characteristics have been defined in a way to cover import technology agnostic properties that have an impact on the maintenance of source code. Thus, most of the technology specific concept and properties can be subordinated to the given Source Code Characteristics. For example, inheritance can be classified as a subcategory of complexity. In addition, an extension of Source Code Characteristics includes to define justified mappings to System Quality Characteristics. Thus, an enlargement of Source Code Characteristics should be avoided. In contrast, Source Code Partitions allow the adaption, if the presented requirements do not fit into the given ones of a specific technology or task. Thus, new ones can be added or removed. It is also eligible to change the categorisation of Source Code Partitions. For example, alternative classification can be provided by a source code's frequency-of-change or usage. Such approaches would additionally justify an adjustment of the rating weights. No restrictions are given for the usage of Source Code Characteristics, as they can be chosen independently. A final summary of the foreseen degree of extensibility for each perspective is given in Table 6.

Perspective	Extensibility	
System Quality Characteristics	$--$	Have to stay unchanged
Source Code Characteristics	$-$	Should stay unchanged
Source Code Partitions	$+$	Can be adapted and reorganised
Sensor Ports	$++$	No restrictions, as chosen individually

Table 6: Summary of the allowed extensibility for each perspective of the presented model.

3.5 Visualisation

The presented model provides different perspectives of maintainability properties, which fit into the needs of various stakeholders. Additionally, the visualisation was created in a way to provide the possibility of showing and hiding different perspectives on demand. Therefore, the initial viewing was designed to only provide a very generalised and abstract overview of the measured results. Similarly, only the results and linkings of the two perspectives of System Quality Characteristics and Source Code Characteristics are shown. This compact visualisation might be enough to give an insight to the customer about existing maintainability issues. In a further step, the perspective of Source Code Partitions can be added. Thus, interim ratings of Source Code Characteristic, in a particular Source Code Partition can be inferred. This allows the observation of Source Code Partitions with maintainability violations. By hovering over the result of a presented Source Code Partition rating, all subordinated Sensor Ports are listed. These can contain further superficial information in the form of a simple description or illustration of the measurement result. Finally, the visualisations allow for adding a link to an external report, which includes detailed information about the results and the application of the Sensor Port.

The final implementation of the visualisation is shown in Figure 7, is publicly available via github.com[2] and was published under the MIT licence. The visualisation is mainly used to present the results of the final project measurements (see chapter 5). It was created as a web application on top of HTML5-boilerplate[3] and implemented mainly by the usage of JQuery[4]. It renders a visualisation of the presented model by using a JavaScript Object Notation (JSON)-File as input, which contains the ratings and measurement details of all applied Sensor Ports.

[2]The corresponding GitHub repository available via github.com/duffleit/Visualisation-of-A-Maintainability-Model-Based-on-Source-Code-Characteristics.
[3]An HTML5 template published under MIT License and available via html5boilerplate.com.
[4]A Javascript library available via jquery.com.

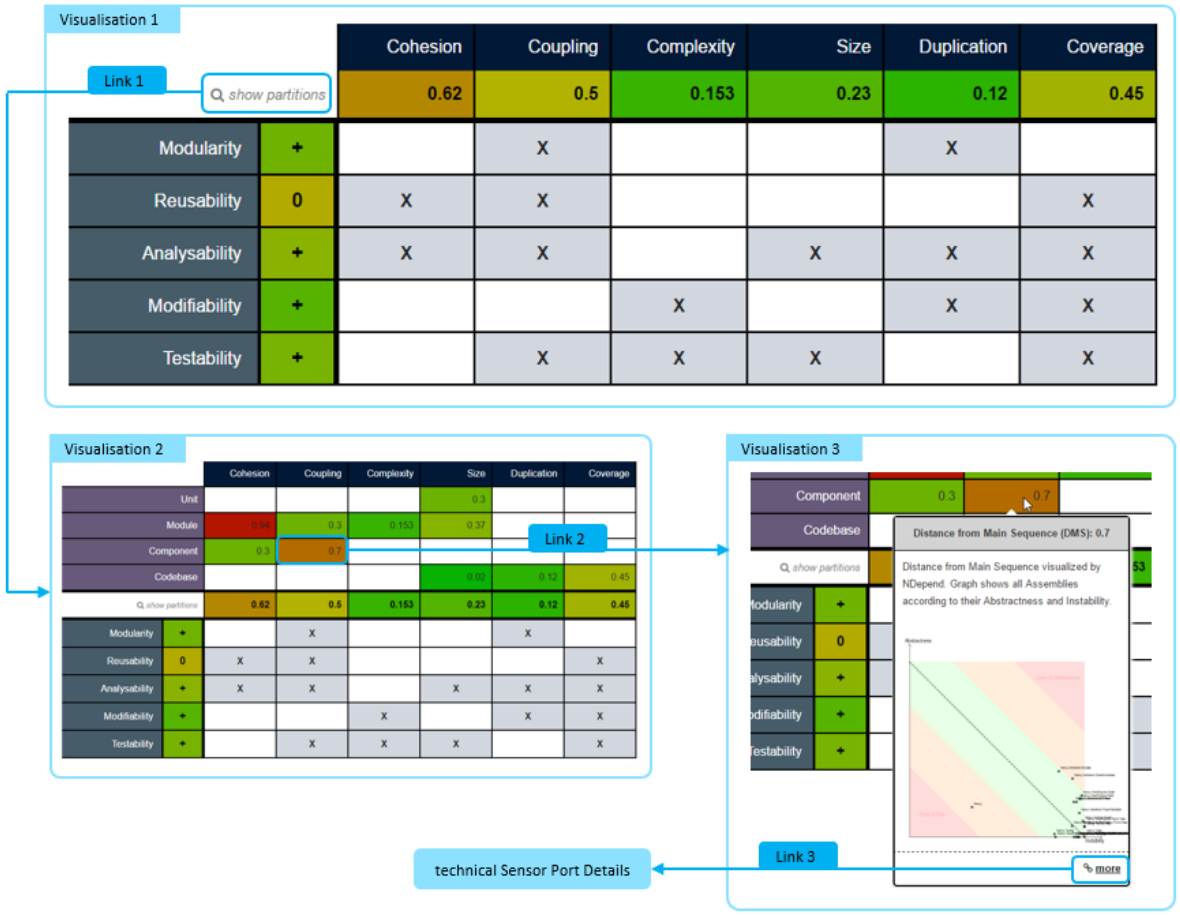

Figure 7: Three different visualisations which contain different perspectives are shown. *Visualisation 1* gives a very compact overview and only displays the linking and results between System Quality Characteristics and Source Code Characteristics. By clicking the button “show partitions” and following *Link 1*, the visualisation is expanded by Source Code Partitions. By hovering over one of the aggregated results and by following *Link 2*, a detailed summary of the Sensor Ports is displayed. Each Sensor Port can provide further technical information that can be displayed by clicking the “more” button and following *Link 3*.

4 Implementation of the Presented Model for C#

The following chapter introduces an implementation for the practical application of the presented model in C#. As explained in chapter 3, the arrangement of Sensor Ports can be chosen depending on the subjected technology. Thus, the selection of Sensor Ports makes up the largest part of this C# implementation.

4.1 Tooling

Various tools have been used to perform the implementation of the presented model in C#. These tools allow the investigation of source code or provide access to relevant data for the measurement. They are briefly presented below:

- **NDepend** is a static code analysis tool, which was already mentioned in section 2.1.2. Since version 4, NDepend comes with a feature called **CQLinq**, to provide a possibility to query .Net projects through Language Integrated Query (LINQ) statements. NDepend contains a default rule-set of over 100 predefined CQLinq statements. These rules are used by NDepend to detect Code Smells and calculate Technical Debt.

  There is also a possibility to define custom rules with CQLinq. This feature was mainly used by the C# implementation of the introduced model, to obtain properties and perform measurements on .NET projects. The created CQLinq rules were implemented by using the *NDepend CQLinq Query Editor* and rely on types of the NDepend-API, which are defined in the namespace `NDepend.CodeModel`. Besides providing access to source code types such as assemblies, types or methods, established metrics are offered out of the box. CQLinq rules can be created by either the query or method syntax of LINQ. For reasons of consistency, all custom rules have been implement by using the query syntax.

- **Visual Studio Enterprise 2017 (VS2017)** comes with different capabilities to observe Source Code Characteristics. Some of these features are (i) the detection of code clones, (ii) the calculation of code metrics for solutions or projects and (iii) observation for code coverage. Furthermore, VS2017 is used to build the investigated .NET projects.

- **DotCover** is a commercial tool distributed by JetBrains[1] that is an extension of VS2017 and calculates statement-level code coverage for .NET applications. DotCover is a good choice to measure the Test Coverage in different codebases, as it supports a variety of test frameworks for .NET such as MSTest, NUnit, or XUnit.

[1] available via jetbrains.com

- **SonarQube** was already described as SQALE implementation (see section 2.1.2) and is used to measure the code duplication for a given codebase.

4.2 Source Code Partition Mappings

The presented Source Code Partitions and their definitions (see section 3.2.1) can be effectively mapped to C# structural concepts. Thus, they can be used without any modifications and are defined in Table 7.

Source Code Partiton		C# structural concept
Unit	$\Longleftrightarrow$	Method
Module	$\Longleftrightarrow$	Class
Component	$\Longleftrightarrow$	Assembly
Codebase	$\Longleftrightarrow$	Solution

Table 7: Mapping between Source Code Partiton and C# structural concepts.

4.3 Sensor Port Implementations

This chapter describes the definition of 12 Sensor Ports that are used by the C# implementation of the model. They are structured by a theoretical overview, the selection of thresholds and their application.

4.3.1 Cyclomatic Complexity

Method

Cyclomatic Complexity (CC) was first defined by McCabe in 1976 [103]. Existing research portrays it as an appropriate measurement for testability and understandability [77]. The metric is based on a programme's control graph and is calculated as

$$M = v(F) = e - n + 2$$

where e defines the edges and n represents the nodes of a given graph F[2]. As a result, the measurement returns the number of linearly independent paths through a method [49, 88]. Kan notes that M can be calculated more simply as the number of decisions made in a method $+1$ [77].

Figure 8 illustrates the control graph of a fictive method. A and D might be decisions like `if`, `for` or `switch` statements. By counting the edges e and the nodes n, the formula can be resolved to $M = v(F) = 7 - 6 + 2$, which results in 3. Again, M can be calculated more simply as the sum of decisions $+1$ $(2 + 1 = 3)$.

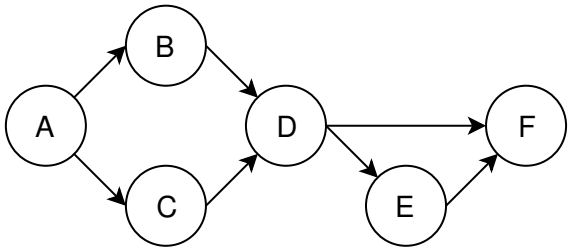

Figure 8: A simple control flow which results in a Cyclomatic Complexity of 3.

The CC metric is additive [77] and can consequently be calculated for different Source Code Partitions such as classes, namespaces or assemblies. In the course of this model, it is only used for methods.

The main criticism for this metric is that statements with a large number of interleaving but excellent readability, such as simple `switch` statements, result in a high CC [110, 128]. Additionally, the different implementations offered by various tools and technologies are criticised [62].

Thresholds

The categorisation and deduction of the rating is done by thresholds introduced by Heitlager et al. [62]. This suggested interpretation is done in three steps.

In the first step, methods are binned into five groups according to certain thresholds shown in Table 8. Next, the relative proportion of the entire codebase is calculated from these groups. This calculation of proportions is based on code lines. For example, to calculate the proportion of methods with high complexity, the sum of code lines that belong to these methods are divided by the sum of all lines in the entire codebase. The final result is deduced from the calculated

[2]McCabe's Cyclomatic Complexity is defined in various other equations such as $v(F) = e - n + p$ or $v(F) = e - n + 2p$. These differences result from different interpretations of graph properties. As p is defined as the number of connected components, it is always equal to 1 for a single method in C#. Therefore, the simplified formula is sufficient and the most accepted form [75, 88]

proportions by using the schema shown in Table 9. This schema defines maximum fractions for each complexity group and thereby determines a result.

CC-Range	Complexity Group
1-10	low complexity
11-20	medium complexity
21-50	high complexity
> 50	very high complexity

Table 8: Thresholds for method grouping used by CC.

medium	high	very high	Rating
< 25%	< 0%	< 0%	0.1
< 30%	< 5%	< 0%	0.3
< 40%	< 10%	< 0%	0.5
< 50%	< 15%	< 5%	0.7
in all other cases			0.9

Table 9: Result schema used by CC to obtain a rating.

Similar thresholds are presented in [159] and [80]. In addition, McCabe himself postulated that a CC of 10 should not be exceeded [77, 103].

Implementation

CQLinq offers a property `CyclomaticComplexity` for methods which is calculated by `1` plus the number of following expressions within a method:

`if, while, for, foreach, case, default, continue, goto, &&, ||, catch, ternary operator (?:), ??;`

The starting point for the calculation is `JustMyCode`. This predefined codebase-view ignores generated and third-party code elements. Subsequently, the sum of all method- and constructor-lines is computed. Methods without CC (e.g. abstract methods) are ignored.

```
1     let types = JustMyCode.Types
2     let NbLines = (double)types
3     .Sum(a => a.MethodsAndContructors
4     .Where(m => m.CyclomaticComplexity.HasValue)
5     .Sum(m => m.NbLinesOfCode)
6     )
```

Code 1: Calculating the sum of all method and constructor code lines

In a further step, the grouping of the methods according to the given thresholds, as defined in Table 8, is carried out. In addition, method-lines are summed and divided by the previously declared `NbLines`.

```
1    let veryHigh = Math.Round((Double)types.Sum(a =>
        a.MethodsAndContructors.Where(m => m.CyclomaticComplexity >
        50).Sum(m => m.NbLinesOfCode))/NbLines,2)
2    let high = Math.Round((Double)types.Sum(a =>
        a.MethodsAndContructors.Where(m => m.CyclomaticComplexity > 20
        && m.CyclomaticComplexity <= 50).Sum(m =>
        m.NbLinesOfCode))/NbLines,2)
3    let medium = types.Sum(a => a.MethodsAndContructors.Where(m =>
        m.CyclomaticComplexity > 10 && m.CyclomaticComplexity <=
        20).Sum(m => m.NbLinesOfCode))/NbLines
```

Code 2: Method grouping and calculation of relative proportions

In closing, the ratings are assigned based on the scheme defined in Table 9. The final `Select` statement returns the calculated rating for the Sensor Port.

```
1    let rating =
2    (medium <= 0.25 && high <= 0 && veryHigh <= 0) ? 0.1 :
3    (medium <= 0.30 && high <= 0.05 && veryHigh <= 0) ? 0.3 :
4    (medium <= 0.40 && high <= 0.10 && veryHigh <= 0) ? 0.5 :
5    (medium <= 0.50 && high <= 0.15 && veryHigh <= 0.05) ? 0.7 : 0.9
6
7    select rating
```

Code 3: Assignment of final ratings

4.3.2 Coupling between Object Classes

Method

Coupling between Object Classes (CBO) is one of six object-oriented design metrics proposed by Chidamber and Kemerer in 1994 [27]. It is defined as the number of relations to other classes [18, 27, 77]. Coupling is caused by method calls, field accesses, inheritance, arguments, return types and exceptions [148].

Confusingly, the counting of couplings is implemented in different ways. These varieties result from ambiguous interpretations [77, 88] of how dependencies are considered due to their direction. Thus, either (i) the direction of the dependencies is not taken into account [18, 48] or (ii) only outgoing dependencies are used for the calculation. The implemented metrics-calculation in VS2017 uses the second approach[3]. The different approaches are illustrated in Table 10.

[3] Observed by the given examples in VS2017.

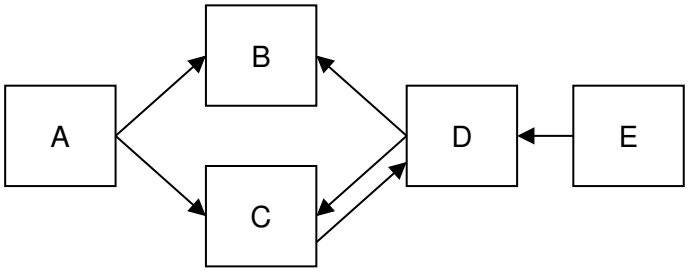

Figure 9: A class diagram with relations to point out the differences between the calculation of a CBO with considering both or only outgoing directions.

	considered directions	
	both (i)	outgoing (ii)
A	2	2
B	2	0
C	2^4	1
D	3	2
E	1	0

Table 10: Results of different CBO under consideration of both or only outgoing directions.

There are also different variants regarding the consideration of inheritance. The proposed version from Chidamber and Kernerer incorporates the coupling by inheritance [28, 43]. CBO', a modified form, ignores it [43].

Different studies indicate a relationship between CBO and the fault-proneness of classes [140]. Therefore, the metric can be used as an indicator of maintainable source code.

Thresholds

Two suggested thresholds for the CBO metric are defined as 9 by [140] and as a range from 0 to 8 by [24]. Based on these examinations, a CBO above 9, which considers inheritance and references in both directions, is interpreted as high. This allows the evaluation of individual classes of a codebase.

In further consequence, the proportion between classes with a high CBO and the number of all classes is calculated. This proportion is determined by using the schema described by Heitlager et al. [62], which has been adapted to two groups and is shown in Table 11. These values are used to provide ratings for CBO within this implementation, as shown in Figure 10.

[4]The couplings of a class in both directions can be interpreted as a set. For this reason, references in both directions are counted only once. The two references between $D \rightarrow E$ and $E \rightarrow D$ are therefore counted as one reference.

Threshold	Rating
$< 25\%$	$++$
$< 35\%$	$+$
$< 50\%$	o
$< 70\%$	$-$
$\geq 70\%$	$--$

Table 11: Thresholds for CBO.

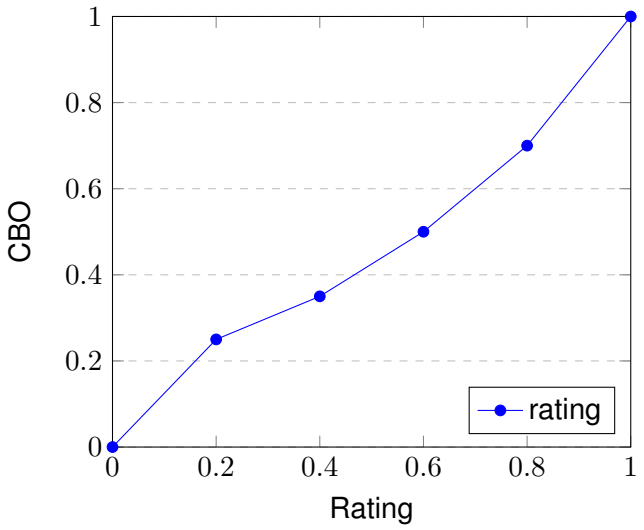

Figure 10: Mapping for a given CBO rating.

Implementation

As VS2017 only offers a modified version of CBO, the metric is reimplemented in CQLinq. The starting point of the calculation is again `JustMyCode`, to ignore third-party code elements.

```
1    let types = JustMyCode.Types
2    let NbTypes = types.Count()
```

Code 4: Measuring the number of types

For further calculation, types are selected whose CBO is higher than the suggested threshold of 8. The CBO is calculated by the sum of `TypesUsed`[5] and `TypesUsingMe`[6]. The `Exclude()` statement enforces that duplicate types are counted only once. The `ExceptThirdParty()` statement is used to filter `TypesUsed`, as system libraries (such as *mscorlib.dll* or *System.dll*) should be ignored.

```
1    let typesWithHighCbo = types.Where(
2    t =>  t.TypesUsed.ExceptThirdParty().Count() +
3    t.TypesUsingMe.Except(t.TypesUsed).Count() > 8)
```

Code 5: Selecting types with high CBO

[5]Property of `Type`, which returns a list of types that are used by the type.
[6]`Property` of Type, which returns a list of types that are using the type.

The final step calculates the proportion of classes with high CBO and derives the rating according to the schema given in Table 11.

```
1     let prop  = typesWithHighCbo.Count() / (double)NbTypes
2
3     let rating =
4     (prop < 0.25) ? prop /(0.25/0.20) :
5     (prop < 0.35) ? (prop - 0.25)/(0.10/0.20) + 0.20 :
6     (prop < 0.50) ? (prop - 0.35)/(0.15/0.20) + 0.40 :
7     (prop < 0.70) ? (prop - 0.50)/(0.20/0.20) + 0.60 :
8     (prop < 1) ? (prop - 0.70)/(0.30/0.20) + 0.80 : 1
9
10    select rating
```

Code 6: Calculated proportions and derived ratings

4.3.3 Distance from Main Sequence

Method

The Distance from Main Sequence (DMS) metric was first defined by Martin in 1994 [100]. It is based on the abstractness and instability of an assembly.

- **Abstractness** measures the proportion of abstract types within an assembly. If an assembly has a high percentage of concrete implementations (such as classes or structures), it is called concrete. In the opposite case, if it largely consists of abstract types (such as interfaces or abstract classes), it is called abstract [85, 148]. Thus, it can be calculated by

$$abstractness(A) = \frac{a}{a+c}$$

  where a defines the number of abstract types and c represents the number of concrete types in a given assembly A. The result describes the abstractness of an assembly A by a number between 0 (a complete concrete assembly) and 1 (an entirely abstract assembly) [85].

- **Instability** measures the difficulty to change an assembly without affecting other assemblies. An interesting observation is that an assembly is referred to as unstable if changes are possible without affecting other assemblies. In the opposite case, it is stable when the change has a large effect. It is calculated by

$$instability(A) = \frac{ec}{ac+ec}$$

where ac (afferent coupling) defines the number of types outside the assembly A which depend on types of A. By implication, ec (efferent coupling) describes the number of types inside A which depend on types outside of A. Equivalent to abstractness, the instability results in a number between 0 (assembly is completely stable) and 1 (assembly is completely unstable). [85, 102]

The Distance from Main Sequence describes the ideal ratio between abstractness and instability. The sum of the two properties should be 1 in the best case. Thus, the more stable an assembly is, the more abstract it should be. Conversely, this means that a very unstable assembly should be very concrete [82, 100]. Figure 11 shows the main sequence to which the assemblies should have the minimum possible distance. Furthermore, the two zones of exclusion are illustrated by: (i) the zone of pain, which indicates assemblies that are difficult to change and hard to extend and (ii) the zone of uselessness, which indicates assemblies that are unnecessarily abstract as they have no dependents [116].

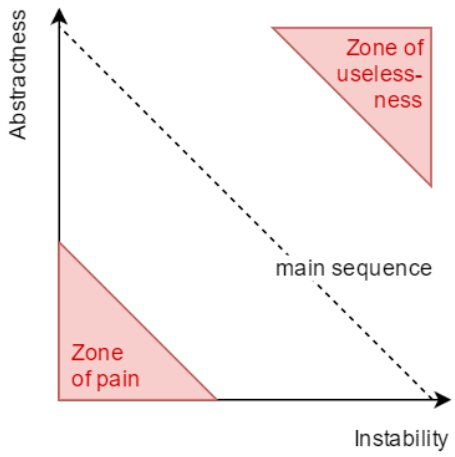

Figure 11: Distance from Main Sequence visualised, including the *zone of uselessness* and the *zone of pain*.

Thresholds

NDepend, which includes an out-of-the-box visualisation for DMS, declares a normalised Distance from Main Sequence higher than 0.7 as problematic [117]. If the value is above 0.3, it is visualised as a warning.

These proposed thresholds are also used for the presented implementation to create proportions of assemblies with high, medium or low DMS. These are calculated by using the number of source code lines. Finally, one of five results for the overall DMS rating is derived by using

the schema described by Heitlager et al. [62], which has been adapted to three groups and is shown in Table 12.

Medium	< 25%	< 30%	< 40%	< 50%	others
High	< 0%	< 5%	< 10%	< 20%	
Rating	0.1	0.3	0.5	0.7	0.9

Table 12: Given thresholds for DMS-proportion and derived ratings.

Implementation

The calculation of DMS is completely performed by the usage of CQLinq. First, the line numbers of non-empty assemblies are summed.

```
1    let assemblies = JustMyCode.Assemblies
2    .Where(a => a.NbLinesOfCode.HasValue && a.NbLinesOfCode.Value > 0)
3    let nbLines = (double) assemblies.Sum(a => a.NbLinesOfCode)
```

Code 7: Measuring the number of assemblies

In the next step, groups are formed based on the given thresholds and their proportions are calculated. CQLinq offers the `NormDistFromMainSeq` property for assemblies, which returns the normalised Distance from Main Sequence in the range of 0 to 1.

```
1    let medium = assemblies.Where(a => a.NormDistFromMainSeq >= 0.3 &&
        a.NormDistFromMainSeq < 0.7).Sum(a => a.NbLinesOfCode) / nbLines
2    let high = Math.Round((double)assemblies.Where(a =>
        a.NormDistFromMainSeq >= 0.7).Sum(a => a.NbLinesOfCode) /
        nbLines,2)
```

Code 8: Calculation of DMS groups and their proportions

In the final step, the calculated proportions are used to return a rating according to the schema given in Table 12.

4.3.4 Relational Cohesion

Method

Martin introduced the Relational Cohesion (RC) metric to represent the average number of internal relationships in an assembly [102]. It is defined as

$$RC(A) = \frac{R(A) + 1}{N(A)}$$

where $R(A)$ describes the sum of all internal relationships[7] for each type of assembly A. $N(A)$ defines the number of classes and interfaces in the observed assembly A. The additional 1 in the formula prevents the result to be 0 if the component only contains a single type ($N(A) = 1$). High RC indicates high cohesion and strongly related types in an assembly. [48, 102]

Figure 12 illustrates a fictive assembly E. It contains four classes (`Person`, `PersonFactory`, `Developer` and `Company`) which have some internal references (`Developer` is a subclass of `Person`; `DeveloperFactory` references `Developer`; `Company` references `Person` and `Developer`)

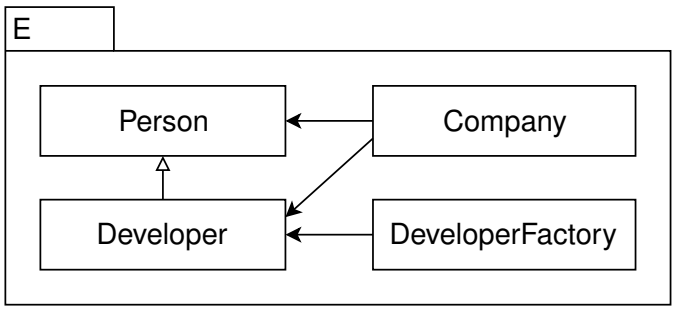

Figure 12: Example of an assembly E with an RC of 1.2.

Thus, $R(E)$ and $N(E)$ are both defined as 4. Consequently the calculated RC of this given example is 1.2.

Thresholds

Martin does not provide any thresholds or indicators for this metric [102]. NDepend suggests 1.5 to 4 as a good range. A value below the specified range indicates that types of an assembly are too weakly related, while a value above indicates that the types are over-coupled. Both cases should be avoided [102, 117]. As suggested for other measurements by Heitlager et al. [62], the final fraction of assemblies with insufficient RC is calculated by source code lines.

Implementation

The measurement for RC is entirely performed with CQLinq. To begin with, all relevant assemblies are collected. NDepend suggests to only use assemblies with more than 20 non-generated types to obtain a meaningful result [117].

[7] relationships to other classes or interfaces within the same assembly

```
1     let relevantAssemblies = JustMyCode.Assemblies.Where(a =>
          a.ChildTypes.Where(t => !t.IsGeneratedByCompiler).LongCount() >
          20)
```

Code 9: Selecting all relevant types

Next, all assemblies which have an RC outside the accepted range are selected. With the help of `TypesUsed` and `Intersect`, a sum of all references within an assembly is calculated, which is then divided by the number of total types. A `HashSet` is used to increase performance.

```
1     let typeLists = relevantAssemblies
2     .Select(a => a.ChildTypes.Where(t =>
          !t.IsGeneratedByCompiler).ToHashSet())
3
4     let assembliesWithHighRc = typeList
5     .Where(tl => (tl.Sum(t => t.TypesUsed.Intersect(tl).Count()) /
          (double)tl.Count) > 4)
6
7     let assembliesWithLowRc = typeList
8     .Where(tl => (tl.Sum(t => t.TypesUsed.Intersect(tl).Count()) /
          (double)tl.Count) < 1.5)
```

Code 10: Selection of assemblies with RC outside the accepted range

Finally, the proportion of assemblies which have an unacceptable RC is calculated by source code lines.

```
1     let nbLinesAssembliesWhereRcIsOutOfRange =
2     assembliesWithHighRc.Sum(tl => tl.Sum(t => t.NbLinesOfCode)) +
3     assembliesWithLowRc.Sum(tl => tl.Sum(t => t.NbLinesOfCode))
4
5     let nbLinesAllAssemblies =
6     typeList.Sum(tl => tl.Sum(t => t.NbLinesOfCode))
7
8     let rating =
          (double)nbLinesAssembliesWhereRcIsOutOfRange/nbLinesAllAssemblies
9
10    select rating
```

Code 11: Calculating proportion based on source code lines

4.3.5 Lack of Cohesion of Methods

Method

Metrics for Lack of Cohesion of Methods (LCOM) exist in different modifications. The naming of these variations is often done by appending numbers (e.g. LCOM1, LCOM2, ...). As this numbering is not always consistent in literature [5, 84, 87, 123, 142], the authors' initials are appended in order to unambiguously identify the LCOM-modifications in this work:

- **LCOM-CK**: Chidamber and Kemerer introduced the first version of the LCOM-metric. It was revised [84] and finally proposed [27] as:

$$LCOM = |P| - |Q|, if |P| > |Q|$$
$$= 0 \; otherwise$$

  For each pair of methods in the observed class, a check is made to determine whether the pair shares common instance variables. If that is the case, Q is increased by one. Otherwise, P is raised by one. If P is larger than Q, the result of LCOM-CK is $P - Q$. For the rest it is 0.

  Consider a class C with three methods (`m, n, o`) and following prerequisites: `m` interacts with instance variables `a, b, c`; `n` interacts with instance variables `b, c, d, e`; `o` interacts with instance variables `f, g, h`.

  Thus, the method pair `mn` uses at least one instance variable in common and Q gets increased. The pairs `mo` and `no` don't have any intersections so P gets raised twice. The LCOM-CK of this class `c` is calculated by 2 minus 1 and therefore results in 1.

  An LCOM-CK of 0 indicates a high cohesive class. If the result is greater than zero, the class tends to consist of logics which are independent of each other and serve different targets. To improve this condition, it should be split into subclasses [27].

- **LCOM-HS:** Henderson-Sellers highlighted that zero could be achieved by a wide range of different cases in LCOM-CK. Even if a high number suggests low cohesion, it cannot be assumed that 0 results in high cohesion. Additionally, they criticised that there is no defined guideline to interpret the results [63]. Thus, they presented a modification of LCOM by normalising the number of methods and variables used by a class [63, 123]:

$$LCOM = \frac{(\frac{1}{a}\sum_{j=1}^{a}\mu(A_j)) - m}{1 - m}$$

This equation uses the number of methods m and attributes a. Additionally, the number of methods which access an attribute are defined by $\mu(A_j)$.

Consider a class `C` with three methods (`m, n, o`) and the following prerequisites: `m` interacts with instance variables `a, b, c`; `n` interacts with instance variables `b, c, d, e`; `o` interacts with instance variables `f, g, h`. For each given instance variable, the number of methods accessing it is counted and finally summed. For the given example, this calculation results in a total number of 10. ($a : 1 + b : 2 + c : 2 + d : 1 + e : 1 + f : 1 + g : 1 + h : 1$). Furthermore, the numerator and denominator result in 1.75 and 2. Thus, the LCOM-HS of class `C` is 0.875.

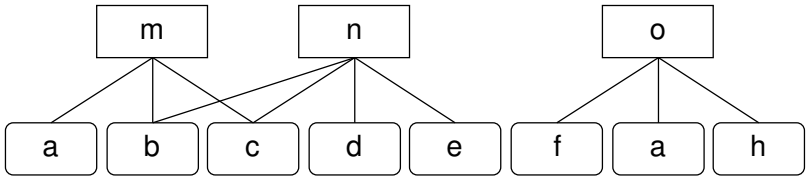

Figure 13: A LCOM-HS example which results in 0.875.

LCOM-HS returns values between 0 and 2. If all methods access all attributes, best-possible cohesion is achieved and the result is 0. If each attribute is only accessed by a single method, worst-possible cohesion is achieved, and the result is 1. A value greater than 1 can only be reached if some attributes are dead, or not used by class internal methods. LCOM-HS returns 0 if there are no methods within the class, and $undefined$ if there are no attributes within the class.

- **LCOM-HM:** Hitz and Montazeri [65] introduced an improved version of the LCOM metric which is based on the graph theory and the grouping of connected methods and attributes. The number of groups is counted and defines the result of LCOM-HM. [65, 127]

  Consider a class `C` with four methods (`m, n, o, p`) and two attributes (`x, y`). The following prerequisites are given: `m` calls `n`; `n` interacts with `x`; `o` and `p` both indicate with `y`;

  This example contains two groups of connected attributes and methods (`mnx`, `opy`) and therefore results in an LCOM-HM of 2. If an interaction from method `o` to attribute `x` is added, the class contains a single group of connected attributes and methods. Thus, the result of LCOM-HM is 1.

  Results higher or equal to 2 indicate a problem, as the class consists of multiple separate groups of connected attributes and methods. In contrast, a result of 1 indicates high cohesion of the class, as all attributes and methods within the class rely on each other [65].

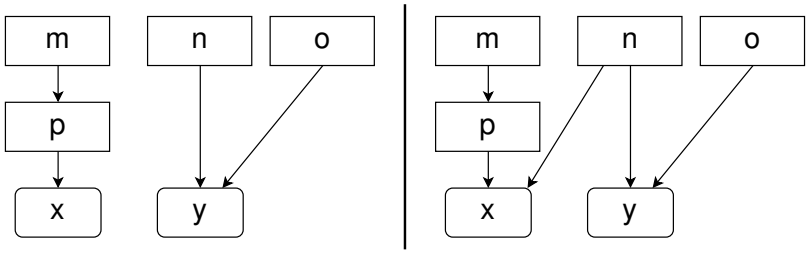

Figure 14: A LCOM-HM examples with two groups (left) and a single group (right).

Due to the well-defined result range, the availability of thresholds and the given tool support in .NET, LCOM-HS is used for further investigation. For the rest of the thesis, LCOM refers to the modified variant of Henderson-Sellers (LCOM-HS).

Thresholds

Filó et al. [51] recommended the usage of thresholds, as shown in Table 13. In order to remain consistent with the previous metrics, the group names are changed from good, regular and bad to high, medium and low. Their analysis is based on Qualitas.*class* Corpus [152], which provides a large number of projects and related measurement results. The LCOM measurement of Qualitas.*class* Corpus is performed by using the modified LCOM-variant from Henderson-Sellers [152].

High	Medium	Low
$lcom \leq 0.167$	$0.167 < lcom \leq 0.725$	$lcom > 0.725$

Table 13: Given thresholds for LCOM.

In order to calculate a valuation for the whole codebase, the schema described by Heitlager et al. [62] is used. It was adapted to three groups as shown in Table 14.

Medium	< 25%	< 30%	< 40%	< 50%	others
High	< 0%	< 5%	< 10%	< 20%	
Result	0.1	0.3	0.5	0.7	0.9

Table 14: Given thresholds for LCOM-proportion and derived ratings.

Implementation

CQLinq is used for measurement, as it comes with out-of-the-box support for LCOM-HS. As preparation for the calculation, types are selected and groups are formed.

```
5     let types = JustMyCode.Types.Where(t => t.LCOMHS.HasValue)
6
7     let medium = types.Where(t => t.LCOMHS.Value  > 0.167 &&
          t.LCOMHS.Value  <= 0.725)
8     let low = types.Where(t => t.LCOMHS.Value  > 0.725)
```

Code 12: Measuring the number of assemblies

Finally, proportions are calculated and the result is derived according to the schema given in Table 14.

```
9     let pMedium = (double)medium.LongCount()/types.LongCount()
10    let pLow = Math.Round((double)low.LongCount()/types.LongCount(),2)
11
12    let rating =
13    (pMedium <= 0.25 && pLow <= 0) ? 0.1 :
14    (pMedium <= 0.30 && pLow <= 0.05) ? 0.3 :
15    (pMedium <= 0.40 && pLow <= 0.10) ? 0.5 :
16    (pMedium <= 0.50 && pLow <= 0.20) ? 0.7 : 0.9
17
18    select rating
```

Code 13: Measuring the number of assemblies

4.3.6 Depth of Inheritance Tree

Method

The Depth of Inheritance Tree (DIT) was introduced by Chidamber and Kemerer [27]. It is described as the maximum length in the inheritance tree from the observed node to its root [27]. Inheritance is one way to achieve reusability in object-oriented languages such as C#. Even if this appears to be beneficial, excessive inheritance leads to the disadvantage of strongly coupled modules and behaviour that is difficult to predict. This leads to structural complexity, which should be kept moderate. [73, 87]

Lanza and Marinescu [87] described a way to calculate the DIT for a complete codebase by calculating the average off all root classes. If a class does not belong to any other class, it is defined as root. Interfaces, framework dependencies or classes from external libraries are not considered. Therefore, this suggested approach can be defined as

$$DIT' = \frac{\sum_{c=1}^{n} DIT_c}{n}$$

where n is the number of root classes and DIT is the length to the deepest subclass of the observed class c [87]. Throughout this thesis, this modified version is called DIT'.

Consider a codebase as shown in Figure 15. It contains four root classes (`A`, `E`, `G`, `I`), as these types do not belong to any other class of the codebase. The calculated depths for these root classes are: $DIT(A) = 2, DIT(E) = 0, DIT(G) = 0, DIT(I) = 1$. Thus, the result of DIT' is calculated as:

$$DIT' = \frac{2 + 0 + 0 + 1}{4} = 0.75$$

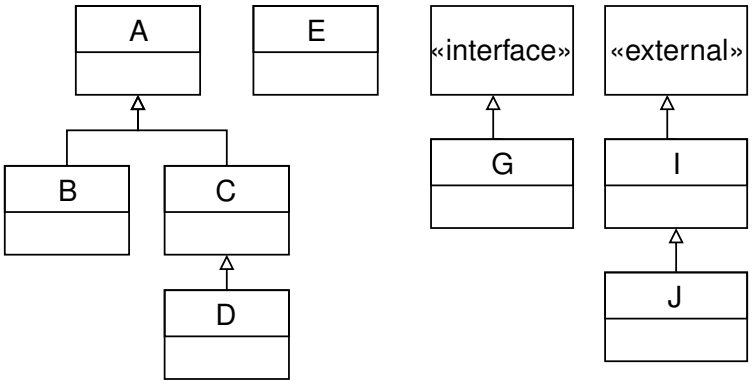

Figure 15: A simple control flow as an example of the calculation of DIT.

Thresholds

Filó et al. [51] suggest the grouping into $dit \leq 2$ (good), $2 < dit \leq 4$ (regular) and $dit > 4$ (bad). Others suggest a maximum of 10 [73]. Lanza and Marinescu [87] provide the thresholds given in Table 15 for their modified version DIT'. These suggested values are used for the described implementation to calculate a result between 0 and 1, as shown in Figure 16. As DIT' does not contain any upper limit, every result ≥ 1 results in a rating of 1.

Threshold	Description
0.09	low
0.21	average
0.32	high

Table 15: Thresholds for DIT'.

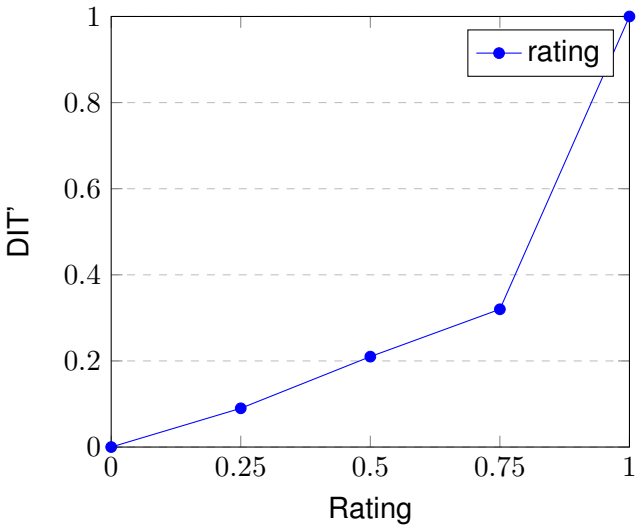

Figure 16: Mappings for a given DIT' rating.

Implementation

CQLinq is used for the implementation of DIT'. All classes which do not derive from any other class within the codebase are selected. `ExceptThirdParty()` ignores classes from external or system libraries (such as *System.Object* or *System.Data.Entity.DbSet*).

```
1     let roottypes = JustMyCode.Types.Where(t => t.IsClass).Where(t =>
          !t.BaseClasses.ExceptThirdParty().Any())
```

Code 14: Selection of all root classes for DIT'

For each root type the maximum length to their deepest derived type is calculated. The calculation results in 0 if a root type has no subclasses. The average of these collected values results in the DIT'.

```
1     let inheritanceDepth = roottypes.Select(t =>
          t.DerivedTypes.Select(dt => dt.DepthOfDeriveFrom(t)).Select(i =>
          i.GetValueOrDefault()).DefaultIfEmpty().Max(i => i)).ToList()
2
3     let dit = inheritanceDepth.Sum(x => x) / (double)roottypes.Count()
```

Code 15: Calculation of DIT'

Finally, a mapping according to schema shown in Figure 16 is performed and the final rating is returned.

4.3.7 Number of Children

Method

Number of Children (NOC) is another metric presented by Chidamber and Kemerer [27]. It is defined as the number of direct subclasses of a given class [73]. Similar to DIT, it also indicates the excessive use of inheritance.

By the heritage of functionality, base-classes have an enormous influence on their direct heirs. Therefore, the number of affected child classes should be limited [73, 87]. Lanza and Marinescu [87] introduced a modified version of NOC which can be applied to a complete codebase. It is defined as

$$NOC' = \frac{\sum_{c=1}^{n} NOC_c}{n}$$

where n is the number of classes and NOC defines the number of direct children for each class [87]. This modified version is called NOC' for the rest of this thesis.

The demonstration of NOC' is again done by considering the codebase shown in Figure 15. It contains a single class (`A`) with two direct children, two classes (`I, C`) with one direct child and five classes (`B, D, E, G, J`) without any children. This leads to the following calculation and result of 0.5.

$$NOC' = \frac{1 * 2 + 2 * 1 + 5 * 0}{8} = 0.5$$

Thresholds

Filó et al. [51] also defined thresholds for NOC[8] as a result of their investigation of existing projects [51]. As for DIT, three categorisations are determined: $NOC \leq 1$ (good), $1 < NOC \leq 3$ (regular) and $NOC > 3$ (bad).

More stringent values are proposed by Lanza and Marinescu [87] for NOC', as shown in Table 16. Based on these values, a rating between 0 and 1 is determined, as illustrated in Figure 17. In the absence of an accepted definition, 1 is used as the upper limit.

[8] The measurements of [51] abbreviate the Number of Children as NSC, as NOC is used as a short form of Number of Classes.

Threshold	Description
0.09	low
0.21	average
0.32	high

Table 16: Thresholds for NOC'.

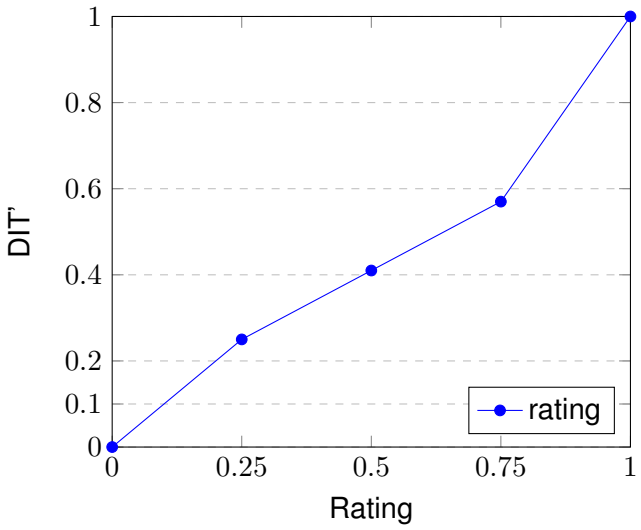

Figure 17: Mapping for given NOC' rating.

Implementation

CQLinq is used to implement NOC'. In a preparatory step, all classes and their direct heirs are selected. By summing all inheritances and dividing them by the number of classes, NOC' can be calculated as follows:

```
1     let classes = JustMyCode.Types.Where(t => t.IsClass)
2     let derivedClasses = classes.Select(c =>
          c.DirectDerivedTypes.Count()).ToList()
3
4     let noc = derivedClasses.Sum(x => x)/(double)classes.Count()
```

Code 16: Calculation of NOC'.

The final rating is calculated by the usage of thresholds explained in Table 16 and Figure 17.

4.3.8 Unit Size

Method

Unit Size (US) is used for measuring the size of methods. Different sources suggested the pure number of code lines as an indicator of maintainability, as it is often correlated with the complexity of a method [62, 157].

Lines of Code: The counting of the Source Line of Code (SLOC or LOC) is not as trivial as one might assume due to a multitude of vague definitions of how the counting is performed [77]. For example, measuring tools treat blank lines, comment lines, or rows that only contain parentheses differently [144]. In addition, a distinction can be made between physical and Logical Line of Code (LLOC) [118].

- **Physical counting** is usually based on the occurrence of carriage returns. Depending on the implementation an optional filtering is performed (e.g. for blank lines, comments, etc.). The advantage is that this type of counting does not need any, or only very trivial, understanding of the programming language [77, 118].
- **Logical counting** is based on the number of logical statements [115]. Counting semicolons is a common way of doing this, although this approach fails in a variety of popular programming languages which do not, or only optionally, use semicolons (SQL, Javascript) [118]. Thus, this type of counting is tied to the respective programming language to ensure the determination of statements [8, 26]. Furthermore, in language-specific definitions, there are differences. A frequently mentioned example is the `for` statement. The result varies between 1 and 3, as it can be considered as a single statement or a construct of an initialisation, a condition and an increment statement [118].

Thus, the definition of counting is defined vaguely in both cases. [115], [77] and [26] indicate logical counting as less dependent on the writing style and more sensitive to the programming language. Therefore, it is considered as more suitable for quality measurements.

Thus, the counting of code lines in this entire thesis is performed by a logical counting based on statements, as demonstrated in Code 17. In detail, a functionality of CQLinq, which counts sequence points (except for brackets) in *PBD*, files is used.

```
1    new List<int>() { 1, 2, 3 }.Where(i => i > 3).Select(i =>
         i.ToString());
2
3    for (int i = 0; i < 100; i++) { i++; }
4
5    while (f < 10) { f++; }
```

Code 17: Examples of SLOC–measurements (3 SLOCs; 4 SLOCs; 2 SLOCs).

Thresholds

Heitlager et al. [62] define five categories for Source Line of Code (SLOC) of methods: < 10 as *simple*, $11 - 20$ as *more complex*, $21 - 50$ as *complex* and > 50 as *untestable*. However, these thresholds are used for the measurements with physical SLOCs that ignore blank and comment

lines. NDepend describes methods with more than 20 logical SLOCs as hard to understand and maintain. Methods with more than 40 logical SLOCs are indicated as extremely complex.

Therefore, the model uses 40 as the top limit. As shown in Table 17, intermediate stages are indicated by 10 and 20. [62] suggests an overall rating by using the same schema as for Cyclomatic Complexity, which is defined in Table 18.

CC-Range	Complexity Group
1-10	simple
11-20	medium
21-50	high
> 40	very high

Table 17: Thresholds for US.

medium	high	very high	Rating
< 25%	< 0%	< 0%	0.1
< 30%	< 5%	< 0%	0.3
< 40%	< 10%	< 0%	0.5
< 50%	< 15%	< 5%	0.7
in all other cases			0.9

Table 18: Mapping of complexity groups to result schema of US.

Implementation

The implementation of this metric is done by using the CQLinq property `NbLinesOfCode`, which returns the number of logical SLOCs. First, all methods that have at least one line of code are used. These are counted for further computation.

```
1    let methods = JustMyCode.Methods.Where(m=>m.NbLinesOfCode.HasValue
        && m.NbLinesOfCode > 0)
2    let nbMethods = (double)methods.Count()
```

Code 18: Selecting all relevant methods

In the final step, groups are formed based on the SLOCs of methods. These are then placed in relation to the number of all methods. These ratios are generalised to a total result by using the schema from Table 18.

```
3    let veryHigh = Math.Round(methods.Count(m => m.NbLinesOfCode >
        40)/nbMethods,2)
4    let high = methods.Count(m => m.NbLinesOfCode > 20 &&
        m.NbLinesOfCode <= 50)/nbMethods
5    let medium = methods.Count(m => m.NbLinesOfCode > 10 &&
        m.NbLinesOfCode <= 20)/nbMethods
6
7    let rating = ...
```

Code 19: Selecting all relevant methods

4.3.9 Codebase Balance

Method

Visser et al. [157] state the importance of a well-balanced software architecture. It is argued that changes of isolated components become simpler if they are of equal size. In contrast, dominant assemblies[9] are affected by a wide range of changes and should therefore be avoided.

Codebase Balance (CBB) is calculated by using the Gini Coefficient [38, 56]. This statistical measure for dispersion was initially introduced to represent distribution of a nation's residents, such as income or wealth [56]. Visser et al. [157] describe it as an appropriate calculation formula for rating the balance between codebases. It is defined in different modifications [38, 165, 166]; the one used for this implementation is defined as

$$G = \frac{1}{n}(n + 1 - 2\frac{\sum_{i=1}^{n}(n+1-i)y_i}{\sum_{i=1}^{n} y_i})$$

where n is the number of all assemblies within a codebase and y is the definition of their size [25, 38]. The size of an assembly is computed by LLOC.

Threshold

Gini results in a range between 0 and 1, where 0 indicates full size-equality components and 1 denotes worst possible distribution. This result range can directly be used as the Sensor Port's rating.

Implementation

As LLOCs are the basis for the calculation, CQLinq was used to implement the measurement. First, all non-testing components with at least a single line of code are selected.

```
1    let NbLinesAssembly = JustMyCode.Assemblies
2    .Where(a => !a.Name.ToLower().Contains("test"))
3    //exlcude testassemblies - logic therefore can be adapted
4    .Where(a => a.NbLinesOfCode.Value > 0)
5    .Select(a => (double)a.NbLinesOfCode.Value).OrderBy(a => a)
6    .ToArray()
```

Code 20: Selecting relevant assemblies for Gini calculation

[9] Dominant assemblies are defined as large assemblies which contain much logic [157].

Finally, all relevant assemblies are counted and the Gini Coefficient is calculated by using the formula described above. The CBB rating is selected.

```
1     let NbAssemblies = NbLinesAssembly.Count()
2
3     let sumNum = Enumerable.Range(1, NbAssemblies)
4     .Sum(i => (NbAssemblies+1-i)*NbLinesAssembly[i-1])
5     let sumDen = Enumerable.Range(1, NbAssemblies)
6     .Sum(i => NbLinesAssembly[i-1])
7
8     let gini =
          (1/(double)NbAssemblies)*(NbAssemblies+1-2*(sumNum/(double)sumDen))
9
10    select gini
```

Code 21: Calculating Gini for given assemblies

4.3.10 Codebase Size

Method

According to Visser et al. [157] the size of a codebase is another important indicator for maintainability. The correlation between volume and maintainability is fairly intuitive, as large software systems tend to consist of larger teams, a more complex design and a longer project duration. These conditions result in reduced quality, lower analysability and harder maintainability [62, 157]. In this thesis, Codebase Size (CBS) is measured by man-years needed to develop a codebase.

Thresholds

The Codebase Size is mostly defined by lines of code or function points. Visser et al. [157] suggest a measurement in language-independent man-years needed to rebuild a codebase. Five thresholds are provided by Heitlager et al. [62] and are shown in Table 19. These are used to calculate a rating illustrated in Figure 18.

Man-years	Rating
< 8	$++$
$8 - 30$	$+$
$30 - 80$	o
$80 - 160$	$-$
> 160	$--$

Table 19: Thresholds for CBS.

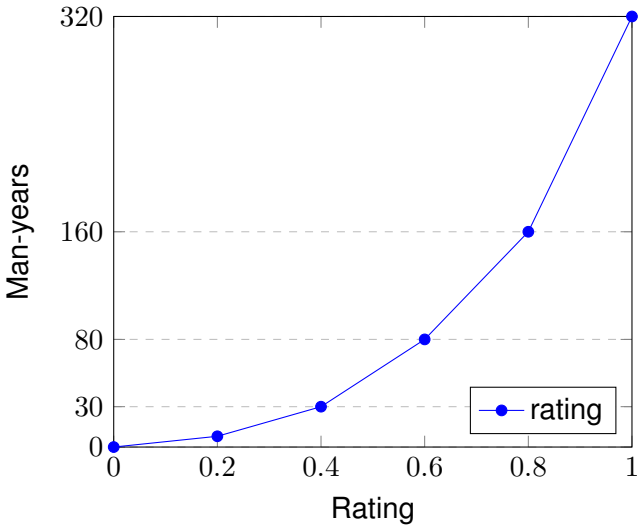

Figure 18: Ratings for a given CBS.

Implementation

CQLinq provides a functionality to calculate the man-years to rebuild a codebase. The NDepend default settings assume 240 workdays per year and that it takes 18 man-days to develop 1000 LLOCs. Thus, the `EffortToDevelop` of self-written source code is transformed to man-years, which are further used to calculate a rating as illustrated in Figure 18. `JustMyCode` is used to ignore third-party code elements.

```
11    let manYears = JustMyCode.Types
12    .Select(e=> e.EffortToDevelop().ToManYear())
13    .Sum(e => e.Value)
14
15    let rating =
16    (manYears < 8) ? manYears/(8/0.2) :
17    (manYears < 30) ? (manYears-8)/(22/0.2) + 0.2 :
18    (manYears < 80) ? (manYears-30)/(50/0.2) + 0.4 :
19    (manYears < 160) ? (manYears-80)/(80/0.2) + 0.6 :
20    (manYears < 320) ? (manYears-160)/(160/0.2) + 0.8 : 1
21
22    select rating
```

Code 22: Calculating Gini for given assemblies

4.3.11 Test Coverage

Method

It is generally accepted in the literature that regression tests have a positive impact on maintainability, as they reduce the introduction of new defects in the case of changes [97]. In addition to increasing reliability, good Test Coverage (TC) is also an indicator for good software testability which is usually caused by good software design [78]. Automated white-box[10] tests are an efficient way to achieve this regression [157]. There are different approaches[11] to measure the coverage of these tests [111]. Heitlager et al. [62] suggest Clover[12], which uses statement coverage, to determine which code is covered by automated tests. There is a wide range of tools that support statement coverage for other languages. For example DotCover, NCover[13] or OpenCover[14] can be used in case of C#. Due to the given tool support, a clearly specified definition [55, 58] and general acceptance in existing research [29, 97], statement coverage is used for this implementation. The rating is determined by the degree by which white-box tests cover the codebase.

Thresholds

Heitlager et al. [62] suggest thresholds for statement coverage, as shown in Table 20. A rating for this implementation, as shown in Figure 19, was derived from these proposed values.

[10] White-box testing (also known as structural testing) is a testing strategy that permits the examination of the internal structure of the subject under test [111].

[11] Myers et al. [111] define them as (i) statement coverage, (ii) decision coverage, (iii) condition coverage, (iv) condition-decision coverage and (v) multiple-condition coverage.

[12] available via atlassian.com/software/clover

[13] available via ncover.com

[14] available via github.com/opencover/opencover

Coverage	Rating
$95 - 100\%$	$++$
$80 - 95\%$	$+$
$60 - 80\%$	o
$20 - 60\%$	$-$
$0 - 20\%$	$--$

Table 20: Thresholds for TC.

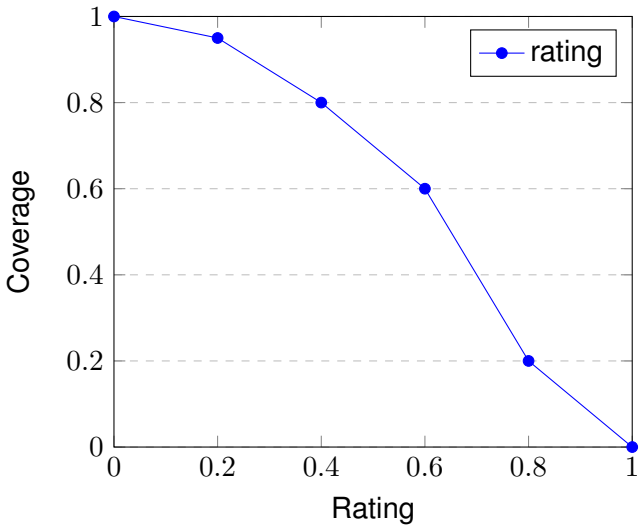

Figure 19: Ratings for a given TC.

Implementation

The measurement is done by the statement coverage provided by DotCover. According to this coverage, a rating is calculated, as shown in Figure 19.

4.3.12 Copy Paste Detection

Method

Duplication is defined as an indicator of maintainability in a variety of literature sources such as [163], [52], [62] or [157]. Duplications (also referred to as Code Clones) can be detected in different ways, such as text-based comparisons [36], token-based approaches [6], the usage of abstract syntax trees [11] or based on dependency graphs [81]. Another taxonomy consisting of four types was presented by Roy et al. [137] and is shown in Table 21.

Type 1	Identical code (except whitespace, layout and comments)
Type 2	Syntactically identical code (except identifiers, literals and types)
Type 3	Syntactically identical code with moved, added or removed statements
Type 4	Code that performs same logic, but is implemented differently

Table 21: Clone types defined by Roy et al. [137]

The existence of tools which use different approaches [168] or a composition of the mentioned types, can be assumed. Some of these strategies are language-independent and can therefore

be generally applied. For example Heitlager et al. [62] suggest detecting language-independent duplications simply by finding equal blocks over six lines (ignoring leading spaces).

As the presented implementation does not have to be language agnostic, a more sophisticated approach can be used. In the context of C#, the code-clones' functionality of VS2017 would be a logical choice, as it detects Type 2 (fully) and Type 3 (partial support) clones that are a minimum of 10 statements long [99]. However, the recommended use cases for this utility are the detection of duplicates during the creation of new code or the modification of existing functionality [99, 146]. Therefore, it acts more as an adviser during implementation instead of providing a comparable rating.

SonarQube provides a Copy Paste Detection (CPD) that is influenced by the Karp-Rabin algorithm [79] and supports Type 2 duplicates, but tends to achieve similar results to the code-clones functionality of VS2017 [121]. In addition it contains good integration into build processes and returns the number of lines that are duplicated, which allows the percentage of duplicates to be calculated. Due to these advantages, SonarQube's implementation is used to perform this measurement.

Thresholds

Eisenberg [40] presents green ($< 5\%$), yellow ($5 - 10\%$) and red ($> 10\%$) status thresholds for the interpretation of SonarQube's CPD. These values are used as basis for the rating of this presented model, which is shown in Figure 20. Duplication $\geq 15\%$ results in a worst-possible rating of 1.

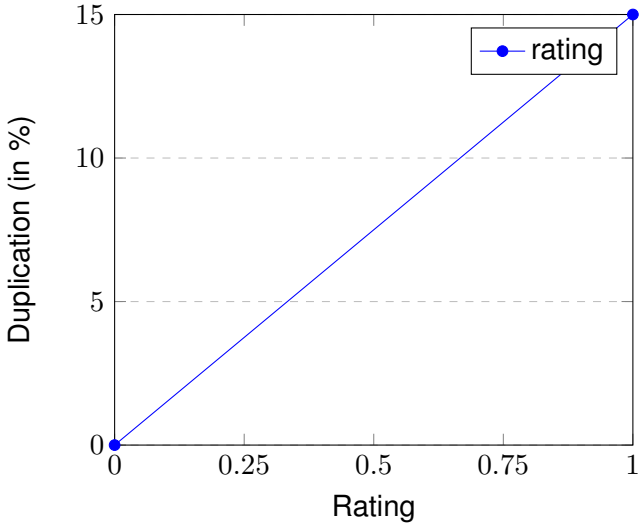

Figure 20: Rating of CPD

Implementation

SonarQube with the scanner for MS-Build was used to measure the percentage of duplicated lines within a codebase. From this result the rating is derived as shown in Figure 20.

4.4 Summary

Sensor Port		*Source Code*	
Metric	Tooling	Characteristics	Partition
Cyclomatic Complexity (CC)	CQLinq	Complexity	Module
Coupling between Object Classes (CBO)	CQLinq	Coupling	Module
Distance from Main Sequence (DMS)	CQLinq	Coupling	Component
Relational Cohesion (RC)	CQLinq	Cohesion	Component
Lack of Cohesion of Methods (LCOM)	CQLinq	Cohesion	Module
Depth of Inheritance Tree (DIT)	CQLinq	Complexity	Module
Number of Children (NOC)	CQLinq	Complexity	Module
Codebase Balance (CBB)	CQLinq	Complexity	Component
Unit Size (US)	CQLinq	Size	Unit
Codebase Size (CBS)	CQLinq	Size	Codebase
Test Coverage (TC)	DotCover	Test Coverage	Codebase
Copy Paste Detection (CPD)	SonarQube	Duplication	Codebase

Table 22: Summary of the 12 Sensor Ports for C# implementation and how they are located in Source Code Characteristics and Source Code Partitions. Additionally, the used tooling for the implementation and measurement is described.

5 Application on Real-World Projects

5.1 Execution

The presented model was applied to different software projects written in C# to provide a possibility for evaluating its results. The source code of these projects has been built by using VS2017. Furthermore, an *NDepend-Project-File* was created for each measured project. Additionally, the 10 Sensor Ports which have been implemented in CQLinq were saved to a single *NDepend-Rule-File*. Both the *NDepend-Project-File* and the *NDepend-Rule-File* have been executed by using the *NDepend-Console* with the following parameters:

```
NDepend.Console.exe <projectname>.nproj /RuleFiles ports.ndrules
```

The remaining two Sensor Ports TC and CPD were measured by using DotCover and SonarQube (with the corresponding SonarQube MS-Build Scanner). The exact versions of all used tools are listed in Table 23. The obtained ratings have been *manually* saved as JSON and are presented by the proposed visualisation (see section 3.5). These final results are compared to maintainability estimations of established approaches and are interpreted on the basis of collected assessments of the measured projects.

Tool	Version
VS2017	15.0.26014.0
NDepend	2017.1.1
SonarQube	6.2
SonarQube MS-Build Scanner	2.2.0.24
DotCover	10.0.2

Table 23: The versions of tools that have been used to execute the Sensor Ports.

5.2 Evaluated Software Projects

The presented model was applied to a set of three open-source and three internal projects. The selection was carried out to offer a wide range of different characteristics, which allows the model's examination from multiple perspectives.

5.2.1 Open-Source Projects

Autofac

Autofac was one of the first open-source Inversion of Control (IOC) containers for .Net [125]. It is a standalone library, with no UI and few dependencies. IOC containers are considered as complex software components since they tend to use a lot of reflection and other sophisticated logics to dissolve required dependencies. Autofac is a very commonly used IOC container in the .NET environment, which is confirmed by a large number of mentions in literature such as [23, 112, 125, 155] and more than 4.3 million downloads via NuGet[1]. The presented model only investigated the core components of Autofac in version 3.5.2.

ShareX

ShareX is open-source software with over 3500 commits and more than 5000 stars on github.com (accessed: 2017-04-06). Thus, it is currently one of the most popular open-source applications in the .NET environment. It allows for capturing and recording screen areas and supports the upload of these images or videos to a variety of different destinations. Moreover, it provides some productivity tools such as a colour picker or monitor tester. The software is therefore a UI-heavy client, which is written mostly by using the Windows Forms Toolkit. The software project was investigated in version 11.5.0.

Nancy

Nancy is a web framework for .Net inspired by Sinatra[2]. It is published under the MIT licence and has over 5000 stars on github.com (accessed: 2017-04-16). In contains a view engine, routing functionalities and different hosting possibilities. It was investigated in two different versions in order to identify possible improvements between two major versions. The two measurements were performed on (i) Nancy version 1.4.3 (named as Nancy-Old), and (ii) Nancy

[1] NuGet is an open-source package manager available via nuget.org.

[2] Sinatra is a web framework originally written in Ruby and has been an inspiration for similar frameworks in other languages such as Spark (Java), Goji (Go-Lang) and Nancy (C#).

version 2.0.0 (named as Nancy-New). Included sample projects have been ignored in both Versions.

5.2.2 Internal Projects

The internal projects are mostly in production and developed for external clients. Thus, they have been anonymised and renamed after brilliant minds in the field of software engineering. Mainly responsible developers were asked about specific characteristics and the maintainability of the selected systems.

Turing

Turing is a web application, that is based on ASP.Net MVC 4. The completion of the initial development was in 2011, but it has been in a continuous maintenance phase since then and is expanded quarterly. The primary task of the application is to organise and edit domain-specific data records. Thus, the application logic is simple and mainly limited to Create, Read, Update, Delete (CRUD) logic. It is known that this application contains many duplications and code clones, as a lot of code was copied and slightly adapted during the initial development. As a result, the behaviour can easily be analysed due to the simple logic, but modifications and changes are costly since they too must be carried out in several places.

Hopper

Hopper is another ASP.net MVC 4 application that was initially created and maintained by a very similar team to that of Turing. The domain of the application is also very similar but avoiding duplication was emphasised during development. Developers perceived a slightly better code quality than in Turing, but the structuring of the application at the assembly level is considered unnecessarily complex, which makes it difficult to customise and extend the software. Compared to Turing, fewer adaptions and extensions were made in Hopper since its completion in 2013.

Torvalds

Torvalds is an application for managing and generating the correspondence of an entire organisation. It was developed from 2006 to 2009 and is based on the Windows Forms Toolkit, Windows Communication Framework (WCF) services and Office Integrations. After the initial development, only minor bug fixes and inevitable changes have been made. The application

is regarded as difficult to maintain due to excessive usage of inheritance and poor cohesion. Following a change in the infrastructure, some components of the application were completely rewritten in 2016, as the absence of tests made changes impossible. Because of this situation, Torvalds was measured three times to allow comparison: (i) the whole legacy application, as it was used in production before adaption (referred to as Torvalds-Complete); (ii) only the legacy components that have been replaced as their maintenance became very costly and error-prone (referred to as Torvalds-Replaced); and (iii) the completely rewritten and newly implemented components with focus on better code quality, test coverage and Maintainability (referred to as Torvalds-New).

5.3 Established Approaches for Comparison

The results of the presented model have been compared to three existing implementations of maintainability measurement methodologies.

Visual Studio flavoured Maintainability Index (VSMI)

The Maintainability Index (MI) provides a measurement method that was applied and proved by a wide range of industrial software systems (see section 2.1.3) [130, 160]. Although it has been criticised by the calculation method [62, 83], it still has a widespread application in software development and is therefore used as a comparative approach. VS2017 is used to calculate this metric, as the implementation does not consider comments. As the presented model does not consider comments either, a more similar foundation is given. Thus, this modified version of the MI fits better, in comparison to the initial approach by Oman and Hagemeister [122]. VS2017 also defines how the MI should be interpreted (see section 2.1.3).

NDepend Maintainability Rating (NDMR)

NDepend offers a lot of functionality to verify the maintainability of software projects. As already mentioned (see section 2.1.2), an overall rating for software system written in C# is one of them. This rating is based on more than 150 code rules [47], and therefore takes many Source Code Characteristics into account that have an influence on the maintainability of a codebase. These comprehensive evaluations, and the fact that the tool is explicitly focused on C# and its specifics, justifies this rating as a good comparative value. NDepend version 2017.1.1 was used and the default settings were not modified.

SonarQube Maintainability Rating (SQMR)

SonarQube is characterised in many cases as an optimal state-of-the-art complement for Continuous Integration (CI) and Continuous Delivery (CD) [22, 80, 129] and has been established in many companies [162]. Due to this currency and prevalence, the debt-based maintainability rating provided for SonarQube is another suitable comparative approach. As with NDepend, no modifications were made to the default settings. SonarQube assesses the maintainability of a software project from A (best-possible maintainability) to E (worst-possible maintainability) and was used in version 6.2, with the MS-Build Scanner in 2.2.0.24.

5.4 Results

The results are presented by the visualisation as explained in section 3.5. Due to limited space, only the first two levels of the model are illustrated. A summary of all projects can be retrieved from Table 30 and detailed results of the used Sensor Ports can be found in Table 31.

Autofac

The result as illustrated in Figure 21 promises good maintainability for Autofac. Above all, the complexity and duplication of the project achieve excellent ratings. In contrast, coverage and coupling are only indicated to be moderate. Since these two properties mainly affect the reusability of a system, this is the only System Quality Characteristic to be assessed as moderate (0). All others are assessed as good ($+$).

		Cohesion	Coupling	Complexity	Size	Duplication	Coverage
show partitions		0.37	0.50	0.15	0.23	0.12	0.45
Modularity	+		X			X	
Reusability	0	X	X				X
Analysability	+	X	X		X	X	X
Modifiability	+			X		X	X
Testability	+		X	X	X		X

Figure 21: Results of the presented model for Autofac.

The comparative measurements differ. While VS2017 and SonarQube evaluate Autofac similarly well (VSMI: 83, SQMR: A), NDepend only achieves a C-rating.

VSMI: 83	$++$	SQMR: A	$++$	NDMR: C	o

Table 24: Results of comparative approaches for Autofac.

ShareX

ShareX results in a moderate maintainability rating, as illustrated in Figure 22. Due to the lack of tests (the ShareX codebase contains no tests), the coverage is evaluated as the worst possible. Coupling and duplication, in contrast, yield very good ratings. This strong divergence between the ratings leads to an average grading of the Software Quality Characteristics, with the exception of modularity, which is only caused by coupling and duplication.

		Cohesion	Coupling	Complexity	Size	Duplication	Coverage
show partitions		0.43	0.18	0.43	0.48	0.09	1.00
Modularity	++		x			x	
Reusability	0	x	x				x
Analysability	0	x	x		x	x	x
Modifiability	0			x		x	x
Testability	0		x	x	x		x

Figure 22: Results of the presented model for ShareX.

The results of the comparative measurements are very similar to those of Autofac. While VS2017 and SonarQube attest very good maintainability (VSMI: 81, SQMR: A), NDepend only evaluates the project with a C-rating.

VSMI: 81	$++$	SQMR: A	$++$	NDMR: C	o

Table 25: Results of comparative approaches for ShareX.

Nancy

The two measurement results of Nancy-Old (illustrated in Figure 23) and Nancy-New (illustrated in Figure 24) are very similar. Improvements have been made in the Source Code Character-

istics of complexity, duplication and cohesion, while coverage has deteriorated. These slight changes have no effect on the System Quality Characteristics, which are equal for Nancy-Old and Nancy-New.

		Cohesion	Coupling	Complexity	Size	Duplication	Coverage
show partitions		0.63	0.34	0.32	0.41	0.04	0.73
Modularity	++		X			X	
Reusability	0	X	X				X
Analysability	0	X	X		X	X	X
Modifiability	+			X		X	X
Testability	0		X	X	X		X

Figure 23: Results of the presented model for Nancy-Old.

		Cohesion	Coupling	Complexity	Size	Duplication	Coverage
show partitions		0.56	0.34	0.18	0.41	0.03	0.79
Modularity	++		X			X	
Reusability	0	X	X				X
Analysability	0	X	X		X	X	X
Modifiability	+			X		X	X
Testability	0		X	X	X		X

Figure 24: Results of the presented model for Nancy-New.

The measurement results provided by VS2017 and NDepend are the same for both projects (VSMI for both: 80, NDMR for both: C). Without modification of the project, SonarQube can only perform the measurement on Nancy-Old (SQMR: A) since the project-type *.xproj*, which is used by Nancy-New, is not yet supported in SonarQube. Thus, the projects of Nancy-New have been changed back to the supported *.csproj* project-type, which results in an A-rating. This modified codebase of Nancy-Old has also been used to measure the TC Sensor Port.

Nancy-Old	VSMI: 80	$++$	SQMR: A	$++$	NDMR: C	o
Nancy-New	VSMI: 80	$++$	SQMR: A	$++$	NDMR: C	o

Table 26: Results of comparative approaches for Nancy Projects.

Turing

The presented model achieves a below-average maintainability for Turing, as illustrated in Figure 25. This result is paralytically caused by the two Source Code Characteristics coverage and duplication, which return a very poor rating. Only coupling can be interpreted as well implemented in this codebase. These weak results mainly affect the System Quality Characteristic modifiability, which is only rated as poor $(-)$. All others are evaluated as moderate (0).

		Cohesion	Coupling	Complexity	Size	Duplication	Coverage
show partitions		0.44	0.25	0.50	0.44	0.75	0.96
Modularity	0		x			x	
Reusability	0	x	x				x
Analysability	0	x	x		x	x	x
Modifiability	-			x		x	x
Testability	0		x	x	x		x

Figure 25: Results of the presented model for Turing.

In contrast to the presented model, the comparative measurements demonstrate good maintainability for Turing. In detail, VSMI results in 81 and an A-rating is calculated by SQMR and NDMR.

VSMI: 81	$++$	SQMR: A	$++$	NDMR: A	$++$

Table 27: Results of comparative approaches for Turing.

Hopper

The measured maintainability of Hopper is comparable to Turing. As illustrated in Figure 26, the Software Quality Characteristics of the codebase are evaluated as moderate (0), except for Reusability, which is considered poor $(-)$. Similar to ShareX, Hopper does not contain any

tests. Thus the Coverage is evaluated as being the worst possible. All other Source Code Characteristics have a good rating, except for coupling, which was rated by 0.66.

		Cohesion	Coupling	Complexity	Size	Duplication	Coverage
show partitions		0.40	0.66	0.26	0.35	0.33	1.00
Modularity	0		X			X	
Reusability	-	X	X				X
Analysability	0	X	X		X	X	X
Modifiability	0			X		X	X
Testability	0		X	X	X		X

Figure 26: Results of the presented model for Hopper.

Similar to Turing, the comparison methods evaluate the maintainability of Hopper in contrast to the presented model. Therefore, VS2017 (VSMI: 84), SonarQube (SQMR: A-rating) and NDepend (NDMR: A-rating) evaluate the maintainability of the Hopper codebase as good.

VSMI: 84	$++$	SQMR: A	$++$	NDMR: A	$++$

Table 28: Results of comparative approaches for Hopper.

Torvalds

The **Torvalds-Complete** codebase, as illustrated in Figure 27, differs within the given Source Code Characteristics. While the modularity is considered very good $(++)$ and the modifiability as good $(+)$, all other characteristics (reusability, analysability, testability) are rated moderate (0). In the case of Source Code Characteristics, duplication and cohesion are emphasised by an alarming rating of 0.79 and 0.58, respectively. The rest of the Source Code Characteristics are measured as good, especially duplication, with a very low value of 0.11.

Based on NDMR, NDepend assigns a B-rating to the maintainability of the codebase. VS2017 and SonarQube come up with a better evaluation, as the VSMI is 78 and the SQMR returns an A-rating.

The **Torvalds-Replace** codebase, illustrated in Figure 28, indicates the worst maintainability of all Torvalds measurements. Compared to Torvalds-Complete, all Source Code Characteristics intersect worse, except size. On the Source Code Characteristics, this has the effect that the evaluation of modularity decreases to good $(+)$. All others remain unchanged.

		Cohesion	Coupling	Complexity	Size	Duplication	Coverage
show partitions		0.58	0.28	0.24	0.39	0.11	0.79
Modularity	++		X			X	
Reusability	0	X	X				X
Analysability	0	X	X		X	X	X
Modifiability	+			X		X	X
Testability	0		X	X	X		X

Figure 27: Results of the presented model for Torvalds-Complete.

		Cohesion	Coupling	Complexity	Size	Duplication	Coverage
show partitions		0.61	0.41	0.36	0.32	0.10	0.72
Modularity	+		X			X	
Reusability	0	X	X				X
Analysability	0	X	X		X	X	X
Modifiability	+			X		X	X
Testability	0		X	X	X		X

Figure 28: Results of the presented model for Torvalds-Replace.

The evaluations by SonarQube and NDepend remain unchanged in comparison to Torvalds-Complete. Only the calculated VSMI increases slightly to 81.

The **Torvalds-New** codebase, illustrated in Figure 29, indicates the best maintainability of all Torvalds measurements. In comparison to Torvalds-Complete and Torvalds-Replace, all Source Code Characteristics have improved. Coverage includes the greatest improvement with a rating of 0.38. On System Quality Characteristics, this has the effect that reusability, analysability, modifiability and testability are rated good ($+$), and modularity very good ($++$).

		Cohesion	Coupling	Complexity	Size	Duplication	Coverage
show partitions		0.53	0.23	0.24	0.27	0.06	0.38
Modularity	++		X			X	
Reusability	+	X	X				X
Analysability	+	X	X		X	X	X
Modifiability	+			X		X	X
Testability	+		X	X	X		X

Figure 29: Results of the presented model for Torvalds-new.

The evaluations by SonarQube and NDepend show no changes. Thus, they achieve a B-rating for NDMR and an A-rating for SQMR. Therefore, SonarQube and NDepend measure Torvalds-Complete, Torvalds-Replace and Torvalds-New equally. VS2017 calculates a VSMI of 78.

Torvalds-Complete	VSMI: 78	$++$	SQMR: A	$++$	NDMR: B	$+$
Torvalds-Replace	VSMI: 81	$++$	SQMR: A	$++$	NDMR: B	$+$
Torvalds-New	VSMI: 78	$++$	SQMR: A	$++$	NDMR: B	$+$

Table 29: Results of comparative approaches for Torvalds Projects.

Codebase	Comparative Approaches			System Quality Characteristics				
	VSMI	NDMR	SQMR	Modularity	Reusability	Analysability	Modifiability	Testability
Autofac	83	C	A	$+$	0	$+$	$+$	$+$
ShareX	81	C	A	$++$	0	0	0	0
Nancy-Old	80	C	A	$++$	0	0	$+$	0
Nancy-New	80	C	A	$++$	0	0	$+$	0
Turing	81	A	A	0	0	0	$-$	0
Hopper	84	A	A	0	$-$	0	0	0
Torvalds-Complete	78	B	A	$++$	0	0	$+$	0
Torvalds-Replace	81	B	A	$+$	0	0	$+$	0
Torvalds-New	78	B	A	$++$	$+$	$+$	$+$	$+$

Table 30: The results from VS2017 (VSMI), NDepend (NDMR) and SonarQube (SQMR) for each evaluated project are compared to the attested System Quality Characteristics (modularity, reusability, analysability, modifiability and testability) of the presented model.

Codebase	Sensor Ports											
	CC	CBO	DMS	RC	LCOM	DIT	NOC	US	CBB	CBS	ATC	CPD
Autofac	0.10	0.30	0.70	0.44	0.30	0.18	0.18	0.30	0.37	0.02	0.45	0.12
ShareX	0.70	0.27	0.10	0.17	0.70	0.20	0.39	0.70	0.59	0.14	1.00	0.09
Nancy-Old	0.70	0.18	0.50	0.75	0.50	0.10	0.15	0.30	0.85	0.08	0.73	0.04
Nancy-New	0.30	0.19	0.50	0.62	0.50	0.10	0.15	0.30	0.86	0.07	0.79	0.03
Turing	0.70	0.40	0.10	0.38	0.50	0.26	0.54	0.30	0.87	0.14	0.96	0.75
Hopper	0.30	0.41	0.90	0.50	0.30	0.13	0.35	0.30	0.72	0.04	1.00	0.33
Torvalds-Complete	0.10	0.47	0.10	0.26	0.90	0.26	0.36	0.30	0.72	0.16	0.79	0.11
Torvalds-Replaced	0.30	0.71	0.10	0.33	0.90	0.34	0.43	0.30	0.63	0.04	0.72	0.10
Torvalds-New	0.10	0.35	0.10	0.55	0.50	0.26	0.37	0.10	0.68	0.03	0.38	0.06

Table 31: All Sensor Ports and their resulting ratings, that have been used to calculate the Source Code Characteristics and System Quality Characteristics for each evaluated project.

5.5 Interpretation

Comparative Approaches

The SQMR by SonarQube and the VSMI by VS2017 result in excellent maintainability properties for all investigated codebases. Thus, all of them are rated with a SQMR A-rating and a VSMI above 78. The NDMR applied by NDepend returns different measurement results. It is striking, that all three open-source projects perform very badly and result in a C-rating. The internal projects Turing and Hopper are evaluated as B and all three Torvalds codebases are considered to have a maintainability of A. Even if the presented model does not return a final aggregated rating, a tendency of maintainability can be made based on the five given System Quality Characteristics. Apart from SQMR and VSMI, which are only of limited use due to their one-sided assessment, the ratings of the presented model also differ strongly from the results given by NDMR. The presented model provides different assessments. The causes of these differences are difficult to identify due to the poor traceability of the existing methodologies.

Software Project Type

The measurement results of internal projects have indicated poor maintainability in comparison to the results of the investigated open-source projects. This corresponds to the investigations given by reports that confirm [33], that software quality tends to be better in open-source projects than in internal (closed-source) projects. In addition, all of the selected internal projects are known for problems in terms of maintainability, except Torvalds-New.

Open-Source Projects

Autofac was the first open-source project that was investigated by the presented model and it was indicated to provide good maintainability. This measurement can be justified by the broad community behind Autofac. For example, the project has over 200 contributors on Github[3] and over 2500 tagged questions on Stack Overflow[4].

The other two open-source projects, ShareX and Nancy deliver similarly good ratings, except for coverage. The community behind Nancy is aware of the need to improve the coverage and is explicitly seeking for help to enhance the tests of the codebase [114]. Furthermore, the evaluation of the two Nancy models result in very similar ratings, which can be explained by research: Thus, only a few changes have been made between the two versions according

[3] according to github.com/autofac/Autofac (accessed: 2017-04-16)
[4] according to stackoverflow.com/questions/tagged/autofac (accessed: 2017-04-16)

to an official blog post of the Nancy project [113] – the main goal was to make the codebase compliant to the CoreCLR, which is the new runtime of .NET Core [112].

Internal Projects

The presented model attests Turing high duplication and poor coverage. As a result, modifiability is particularly impaired. The remaining properties are considered as moderate. These evaluations fit very well with the presumptions of the responsible developers. Thus, the analysis of the system is more easily possible than carrying out changes. In addition, the existing duplication was successfully identified by the model.

Hopper and Turing result in similar measuring results. Differences exist in duplication, which has diminished, and coupling, which has increased. The first change can be explained by the fact that this was explicitly taken into account during development. The second change can be attributed to the suspicion of the interviewed developers that the structuring at assembly level is unnecessarily complicated. A closer look at the DMS Sensor Port, which measures the coupling at the component level, confirms this assumption, as it returns a rating of 0.8. This example illustrates the advantage of the given root-cause analysis in the presented model.

Among all measured projects of Torvalds, Torvalds-Replace results in the worst ratings of all Source Code Characteristics – except for size (which is indicated by the Sensor Port rating of CBS). From a management point of view, it can be reasoned that replacing Torvalds-Replace with Torvalds-New was the right decision. The fact that the results of Torvalds-Complete are only slightly better than those of Torvalds-Replace support the assumption that the whole codebase is very difficult to maintain. The comparison between Torvalds-Replace and Torvalds-New confirms that the redevelopment of these components, in which much emphasis was placed on better code quality and test coverage, can be regarded as successful since both the System Quality Characteristics and the Source Code Characteristics, have improved enormously.

6 Discussion

Comparison with Existing Approaches

The performed measurements on real-world projects provided results that fit the assumptions of interviewed developers remarkably well. Nevertheless, great differences were found in comparison to existing approaches, but also within these comparative approaches themselves. Predictably, all four approaches (the one presented in this thesis, plus the three chosen for comparison) yielded different results. For this reason, I propose the following three parameters which have a strong influence on the obtained results of existing maintainability measurements: (i) the choice of metrics or rules, (ii) the definition of thresholds, and (iii) the aggregation of results.

The first one is trivial: only measured properties can influence the measurement result. As an example, SonarQube does not offer a measurement for cohesion, which is why this property is not included in the calculated maintainability rating. This fact results in measurement differences with models which consider cohesion. Hence, a broad spectrum of metrics was used in the presented model.

The second mentioned parameter is the selection of thresholds which is significant for the result. In other words, each change of the used thresholds will inevitably result in a different measurement result. Accordingly, established thresholds of existing investigations have been reused in the presented model.

Finally, the calculation of an aggregated result also has a big impact on the resulting maintainability rating of a model. For example, individually weighted interim results have a different impact on the final result. Good examples are Debt-based Approaches in which each vulnerability is rated by a debt. The selection of these debts is essential to the final result. In particular, the justification of such individual weightings is highly disputable and adds complexity to the model; moreover, their visualisation is a particular challenge. Given these constraints, the presented model does not use weightings and calculates the average to aggregate results.

In summary, all these parameters indicate that it is essential to know and understand the applied measurement. It is pointless to perform a measurement without understanding the used metrics, the used thresholds and the aggregation of results. The absence of this knowledge makes the interpretation of results impossible and the gained ratings worthless. The presented model improves this weakness by providing different perspectives and a sophisticated root-cause analysis.

Structure of the Presented Model

Instead of a generalised view, the model consists of several layers which cover different responsibilities for various stakeholders. The choice of System Quality Characteristics proved to be very helpful in practice. As also observed by others [62], these characteristics provide a good foundation for discussions between different stakeholders and to create a common understanding of the maintainability of a software system.

Less trivial was the definition of appropriate Source Code Characteristics. The aim was to limit the number of these attributes because each additional Source Code Characteristic complicates the linking to System Quality Characteristics and thereby impairs traceability. Accordingly, the chosen Source Code Characteristics were defined very abstractly, to effectively classify the properties which affect software maintainability. Inheritance, for example, is only feasible for object-oriented languages and can be classified as a form of complexity. Other examples are dependencies (which can be subordinated to coupling) and structural complexity (which can be subordinated to complexity).

The proportion of comments is often considered as significant for the maintainability of a software project. The empirical information collected from measurements in real-world projects shows that comments can have an opposite effect. They often describe a problem in source code and can be considered as *documented Technical Debts*. In addition, a semantic interpretation would be essential for a meaningful examination of comments, as many are autogenerated, are outdated or consist of commented-out source code [62]. For this reason, comments have not been considered in the presented model. However, multiple possibilities are given if they are assessed as important: (i) comments can be subordinated to an existing Source Code Characteristic such as complexity (as good documentation decreases complexity) or (ii) the model can be extended by a new Source Code Characteristic – for example one named code-documentation.

The previous example illustrates that the well-defined extensibility of the model presents a great advantage when making adjustments to the given requirements. Extensions can be performed on different levels: (i) for all executed measurements, (ii) for measurements in a specific technology, or (iii) for a specific project. The more specific the level of an extension, the worse the comparability of the measured results. Correspondingly, this trade-off should be well respected and extensions should only be applied if urgently necessary or if the aspect of comparability can be neglected.

Tradeoffs in Measuring Maintainability

There are some aspects that have not been considered in the presented model, such as source code parts that have different levels of importance. Johnson [74] from the Standish Group Study Reports reported that two-thirds of software functions are rarely or never used. Only 20% are often or always used – the rest are categorised as “sometimes used”. Different reports [61, 158] confirm that this different distribution of usage correlates with the different frequency of change. This means that the relevance of maintainability differs in different parts of the source code therefore, this weighting should also be included in the final result. The presented model provides two options to address this inclusion: (i) each Sensor Port handles the source code parts of different importance in isolation, or (ii) the Source Code Partitions are restructured to represent these different levels of importance in the source code. Both approaches are possible through the extensibility of the presented model.

Another aspect can be traced back to a lack of understanding of the source code semantics. In this regard, Evans [45] indicated that better the correlation between the domain and business model, better the maintainability of a software system. This is justified by the fact that poor correlation between these two models increases the effort needed to apply changes in the business model to the software system. This aspect was not considered in the present model or any other evaluated approach, mainly because of a poor understanding of the source code semantics, as this is the only basis to compare the domain and business models. Because of a lack of semantic understanding, the aspect of naming in source code can only be assessed in a very limited manner; consequently, any kind of semantic has been ignored in the presented model.

Furthermore, the restriction to static analysis can be listed as a missing aspect. Thus, dynamic analysis – like the validation of useful logging during execution – can additionally be used as an indicator of software maintainability. These or other dynamic subjects were not considered in the course of this thesis and can be regarded as out of scope.

Maintainability is related to many aspects around software development. Therefore it has to be mentioned, that the present model was only intended to focus on Source Code Characteristics, hence it does not address aspects such as usage, semantic or dynamic analysis.

7 Conclusion

This thesis was introduced with a quotation of Tom DeMarco [35], who recognised that we cannot control things which we cannot measure. At the end of this thesis, I am still convinced that this assumption can be directly mapped to the topic of software maintainability: if we have no possibility of figuring out the maintainability of software systems, we will never know if we are moving into the right direction. Nevertheless, I think there is a common misunderstanding of the meaning of “measuring something”. Most existing models mostly indicated this with an easy-to-interpret and comparable result, but having a single number is not relevant for obtaining insight into the maintainability of a software system. Maintainability is a set of different characteristics that cannot be aggregated meaningfully into a single number. Therefore, the presented model deliberately avoided a final maintainability rating and results in various outcomes for different perspectives. Compared to other existing models, this brings the decisive advantage that the measured results can be more easily analysed and interpreted. This fact, in combination with the available linking which enables root-cause analysis, provides a common understanding for maintainability issues and their causes. Furthermore, I understood the importance of using a well-established terminology. Thus, the presented model is based on the characteristics of the ISO/IEC 25010 standard, which supports the discussion on maintainability by different stakeholders. Moreover, a hybrid approach between being dependent on a specific technology and providing a completely generalised methodology was chosen. This brings together the advantages of these different approaches. Finally, the reliability of the presented model was proven by measuring real-world projects and discussing it in comparison with existing approaches for measuring software maintainability. In summary, this thesis introduces a model that improves the measurement and analysis of maintainability by enabling root-cause analysis and providing different perspectives for various stakeholders.

8 Future Work

Future work will focus on applying the presented model to different types of programming languages. Thus, it is important to evaluate whether the used terminology and characteristics of different perspectives match functional or procedural language types in a meaningful way. The measurement by these different programming languages should also be integrated into existing CI processes. Thus, an implementation of the presented model has to be provided for different CI solutions such as Jenkins or Team Foundation Server. In another step, a plugin which provides the visualisation of the model for NDepend and SonarQube would be conceivable. Especially SonarQube is an interesting opportunity, as it offers to define new rules by Roslyn analysers. In the long run, this open source alternative should end the dependency on the commercial NDepend in the .NET implementation.

Another area of activity is the variation of Source Code Partitions and their evaluation. Thus, it would be conceivable to choose the partitions based on their maintenance susceptibility. Parts of the software system which are often affected by maintenance are located in a different partition from parts that are barely concerned. This clear grouping adds a different perspective of the investigated software projects. A similar approach is to structure the Source Code Partitions by components. Each component would be a separate partition, which has the advantage that the maintainability is individually verifiable for each component. I am looking forward to providing different forms of the model, which differ in the definition and structure of source code partitions.

Bibliography

[1] F. Brito e Abreu and Walcelio Melo. “Evaluating the impact of object-oriented design on software quality”. In: *Software Metrics Symposium, 1996., Proceedings of the 3rd International*. IEEE, 1996, pp. 90–99. (Visited on 2017-04-10).

[2] Krishan K. Aggarwal, Yogesh Singh, and Jitender Kumar Chhabra. “An integrated measure of software maintainability”. In: *Reliability and maintainability symposium, 2002. Proceedings. Annual*. IEEE, 2002, pp. 235–241. (Visited on 2017-04-21).

[3] Sheikh Fahad Ahmad, Mohd Rizwan Beg, and Mohd Haleem. “A Comparative Study of Software Quality Models”. In: *International Journal of Science, Engineering and Technology Research* 2.1 (2013), p. 172. (Visited on 2017-01-30).

[4] Robert S Arnold. “Software reengineering: a quick history”. English. In: *Communications of the ACM* v37.n5 (May 1994), p13(2).

[5] Linda Badri and Mourad Badri. “A Proposal of a new class cohesion criterion: an empirical study”. In: *Journal of Object Technology* 3.4 (2004), pp. 145–159. (Visited on 2017-01-24).

[6] Brenda S. Baker. “On finding duplication and near-duplication in large software systems”. In: *Reverse Engineering, 1995., Proceedings of 2nd Working Conference on*. IEEE, 1995, pp. 86–95. (Visited on 2017-03-05).

[7] Tibor Bakota et al. “A probabilistic software quality model”. In: *Software Maintenance (ICSM), 2011 27th IEEE International Conference on*. IEEE, 2011, pp. 243–252. (Visited on 2017-04-08).

[8] N. Balaji, N. Shivakumar, and V. Vignaraj Ananth. “Software cost estimation using function point with non algorithmic approach”. In: *Global Journal of Computer Science and Technology* 13.8 (2013). (Visited on 2017-02-18).

[9] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. en. Addison-Wesley, Sept. 2012. ISBN: 978-0-13-294278-2.

[10] Kathrin Baumann. *Unterstützung der objektorientierten Systemanalyse durch Softwaremaße: Entwicklung eines meßbasierten Modellierungsratgebers*. de. Springer-Verlag, July 2013. ISBN: 978-3-662-13273-9.

[11] Ira D. Baxter et al. “Clone detection using abstract syntax trees”. In: *Software Maintenance, 1998. Proceedings., International Conference on*. IEEE, 1998, pp. 368–377. (Visited on 2017-03-05).

[12] P. Bengtsson and Jan Bosch. “Architecture level prediction of software maintenance”. In: *Software Maintenance and Reengineering, 1999. Proceedings of the Third European Conference on*. IEEE, 1999, pp. 139–147. (Visited on 2017-04-24).

[13] K. Bennett. “An overview of maintenance and reverse engineering”. In: *The REDO compendium*. John Wiley & Sons, Inc., 1993, pp. 13–34. (Visited on 2017-04-24).

[14] James M. Bieman and Byung-Kyoo Kang. “Cohesion and Reuse in an Object-oriented System”. In: *Proceedings of the 1995 Symposium on Software Reusability*. SSR ’95. New York, NY, USA: ACM, 1995, pp. 259–262. ISBN: 978-0-89791-739-1. (Visited on 2017-01-27).

[15] Barry W. Boehm. “The high cost of software”. In: *Practical Strategies for Developing Large Software Systems* (1975), pp. 3–15.

[16] Barry W. Boehm, John R. Brown, and Mlity Lipow. “Quantitative evaluation of software quality”. In: *Proceedings of the 2nd international conference on Software engineering*. IEEE Computer Society Press, 1976, pp. 592–605. (Visited on 2017-01-31).

[17] G. Boetticher, Kankanahalli Srinivas, and David A. Eichmann. “A neural net-based approach to software metrics”. In: (1992). (Visited on 2017-04-30).

[18] Lionel C. Briand, John W. Daly, and Jurgen K. Wust. “A unified framework for coupling measurement in object-oriented systems”. In: *IEEE Transactions on software Engineering* 25.1 (1999), pp. 91–121. (Visited on 2017-02-12).

[19] Frederick P Brooks Jr. *The Mythical Man-Month, Anniversary Edition: Essays On Software Engineering*. en. Pearson Education, Aug. 1995. ISBN: 978-0-13-211916-0.

[20] Manfred Broy et al. “What characterizes a (software) component?” In: *Software-Concepts & Tools* 19.1 (1998), pp. 49–56. (Visited on 2016-12-30).

[21] Gianluigi Caldiera and Victor R. Basili. “Identifying and qualifying reusable software components”. In: *Computer* 24.2 (1991), pp. 61–70. (Visited on 2017-03-30).

[22] G. Ann Campbell and Patroklos P. Papapetrou. *SonarQube in Action*. en. Manning, 2013. ISBN: 978-1-61729-095-4.

[23] James Chambers, David Paquette, and Simon Timms. *ASP.NET Core Application Development: Building an application in four sprints*. en. Microsoft Press, Nov. 2016. ISBN: 978-1-5093-0409-7.

[24] E. Chandra and P. Edith Linda. “Class break point determination using CK metrics thresholds”. In: *Global journal of computer science and technology* 10.14 (2010). (Visited on 2017-02-12).

[25] Chau-Nan Chen, Tien-Wang Tsaur, and Tong-Shieng Rhai. “The Gini coefficient and negative income”. In: *Oxford Economic Papers* 34.3 (1982), pp. 473–478. (Visited on 2017-02-08).

[26] Yue Chen et al. “An empirical study of eServices product UML sizing metrics”. In: *Empirical Software Engineering, 2004. ISESE'04. Proceedings. 2004 International Symposium on*. IEEE, 2004, pp. 199–206. (Visited on 2017-02-18).

[27] Shyam R. Chidamber and Chris F. Kemerer. “A metrics suite for object oriented design”. In: *IEEE Transactions on software engineering* 20.6 (1994), pp. 476–493. (Visited on 2017-01-24).

[28] Shyam R. Chidamber and Chris F. Kemerer. *Towards a metrics suite for object oriented design*. Vol. 26. 11. ACM, 1991. ISBN: 0-201-55417-8. (Visited on 2016-12-30).

[29] John Joseph Chilenski and Steven P. Miller. “Applicability of modified condition/decision coverage to software testing”. In: *Software Engineering Journal* 9.5 (1994), pp. 193–200. (Visited on 2017-03-05).

[30] Michael Jesse Chonoles and James A. Schardt. *UML 2 For Dummies*. en. John Wiley & Sons, Apr. 2011. ISBN: 978-1-118-08538-7.

[31] Don Coleman et al. “Using metrics to evaluate software system maintainability”. In: *Computer* 27.8 (1994), pp. 44–49. (Visited on 2017-01-31).

[32] *Compute and Manage the Technical Debt with NDepend*. URL: http://www.ndepend.com/docs/technical-debt (visited on 2017-04-29).

[33] Coverity. *Coverity Open Source Report 2014*. 2014. URL: http://go.coverity.com/rs/157-LQW-289/images/2014-Coverity-Scan-Report.pdf (visited on 2106-04-16).

[34] Ward Cunningham. “The WyCash Portfolio Management System”. In: *SIGPLAN OOPS Mess.* 4.2 (Dec. 1992), pp. 29–30. ISSN: 1055-6400.

[35] Tom DeMarco. *Controlling Software Projects: Management, Measurement & Estimation*. en. Yourdon Press, 1982. ISBN: 978-0-917072-32-1.

[36] Rahadian Dustrial Dewandono, Fahmi Akbar Saputra, and Siti Rochimah. *Clone detection using Rabin-Karp parallel algorythm*. Department of Informatics, InstitutTeknologiSepuluhNopember, Surabaya, 2013. (Visited on 2017-03-05).

[37] Harpal Dhama. “Quantitative models of cohesion and coupling in software”. In: *Journal of Systems and Software* 29.1 (1995), pp. 65–74. (Visited on 2017-01-02).

[38] Robert Dorfman. “A Formula for the Gini Coefficient”. In: *The Review of Economics and Statistics* 61.1 (1979), pp. 146–149. ISSN: 0034-6535. (Visited on 2017-03-04).

[39] R. Geoff Dromey. “A model for software product quality”. In: *IEEE Transactions on software Engineering* 21.2 (1995), pp. 146–162. (Visited on 2017-02-01).

[40] Robert J. Eisenberg. “A threshold based approach to technical debt”. In: *ACM SIGSOFT Software Engineering Notes* 37.2 (2012), pp. 1–6. (Visited on 2017-03-06).

[41] Thomas J. Emerson. “A discriminant metric for module cohesion”. In: *Proceedings of the 7th international conference on Software engineering*. IEEE Press, 1984, pp. 294–303. (Visited on 2017-04-30).

[42] Thomas J. Emerson. “Program testing, path coverage, and the cohesion metric”. In: *Proc. of the 8th Annual Computer Software and Applications Conference*. 1984, pp. 421–431.

[43] Hakan Erdogmus and Oryal Tanir. *Advances in Software Engineering: Comprehension, Evaluation, and Evolution*. en. Springer Science & Business Media, Mar. 2013. ISBN: 978-0-387-21599-0.

[44] Duggan Evan. *Measuring Information Systems Delivery Quality*. en. Idea Group Inc (IGI), Mar. 2006. ISBN: 978-1-59140-859-8.

[45] Eric Evans. *Domain-driven Design: Tackling Complexity in the Heart of Software*. en. Addison-Wesley Professional, 2004. ISBN: 978-0-321-12521-7.

[46] Giovanni Falcone. *Hierarchy-aware software metrics in component composition hierarchies*. Logos Verlag Berlin GmbH, 2010. ISBN: 978-3-8325-2568-2. (Visited on 2017-02-01).

[47] *Features*. URL: http://ndepend.com/features (visited on 2017-04-29).

[48] Norman Fenton and James Bieman. *Software Metrics: A Rigorous and Practical Approach, Third Edition*. en. CRC Press, Oct. 2014. ISBN: 978-1-4398-3823-5.

[49] Norman Fenton, Shari Lawrence Pfleeger, and Robert L. Glass. “Science and substance: A challenge to software engineers”. In: *IEEE software* 11.4 (1994), pp. 86–95. (Visited on 2017-01-27).

[50] Roy Thomas Fielding. “Architectural styles and the design of network-based software architectures”. PhD thesis. University of California, Irvine, 2000. (Visited on 2017-03-31).

[51] T. G. S. Filó, M. A. da Silva Bigonha, and K. A. M. Ferreira. “A catalogue of thresholds for object-oriented software metrics”. In: *The First Int. Conf. on Advances and Trends in Software Engineering*. 2015, pp. 48–55.

[52] Martin Fowler, K. Beck, and W. Roberts Opdyke. “Refactoring: Improving the design of existing code”. In: *11th European Conference. Jyväskylä, Finland*. 1997. (Visited on 2017-04-10).

[53] Martin Fowler and Jim Highsmith. “The agile manifesto”. In: *Software Development* 9.8 (2001), pp. 28–35. (Visited on 2017-03-29).

[54] Hamido Fujita and Imran Zualkernan. *New Trends in Software Methodologies, Tools and Techniques*. en. IOS Press, 2008. ISBN: 978-1-58603-916-5.

[55] Jerry Gao, H.-S. J. Tsao, and Ye Wu. *Testing and Quality Assurance for Component-based Software*. en. Artech House, 2003. ISBN: 978-1-58053-735-3.

[56] C. Gini. “Concentration and dependency ratios”. In: *Rivista di politica economica* 87 (1997), pp. 769–792.

[57] Robert B. Grady. *Practical software metrics for project management and process improvement*. Prentice-Hall, Inc., 1992. ISBN: 0-13-720384-5. (Visited on 2017-04-09).

[58] Dorothy Graham, Erik Van Veenendaal, and Isabel Evans. *Foundations of Software Testing: ISTQB Certification*. en. Cengage Learning EMEA, Jan. 2008. ISBN: 978-1-84480-989-9.

[59] Penny Grubb and Armstrong A. Takang. *Software Maintenance: Concepts and Practice*. en. World Scientific, 2003. ISBN: 978-981-238-426-3.

[60] Maurice Howard Halstead. *Elements of software science*. Vol. 7. Elsevier New York, 1977. ISBN: 0-444-00205-7. (Visited on 2017-04-08).

[61] Matthew S. Harrison and Gwendolyn H. Walton. "Identifying high maintenance legacy software". en. In: *Journal of Software Maintenance and Evolution: Research and Practice* 14.6 (Nov. 2002), pp. 429–446. ISSN: 1532-0618. (Visited on 2017-05-02).

[62] Ilja Heitlager, Tobias Kuipers, and Joost Visser. "A practical model for measuring maintainability". In: *Quality of Information and Communications Technology, 2007. QUATIC 2007. 6th International Conference on the*. IEEE, 2007, pp. 30–39. (Visited on 2016-12-20).

[63] Brian Henderson-Sellers, Larry L. Constantine, and Ian M. Graham. "Coupling and cohesion (towards a valid metrics suite for object-oriented analysis and design)". In: *Object oriented systems* 3.3 (1996), pp. 143–158.

[64] Jim Highsmith. "Zen and the art of software quality". In: *Agile2009 conference*. 2009. (Visited on 2017-04-10).

[65] Martin Hitz and Behzad Montazeri. "Measuring coupling and cohesion in object-oriented systems". In: (1995). (Visited on 2017-01-24).

[66] Raghu V. Hudli, Curtis L. Hoskins, and Anand V. Hudli. "Software metrics for object-oriented designs". In: *Computer Design: VLSI in Computers and Processors, 1994. ICCD'94. Proceedings., IEEE International Conference on*. IEEE, 1994, pp. 492–495. (Visited on 2017-03-30).

[67] I. IEC. "ISO/IEC 25000 software engineering software product quality requirements and evaluation (SQuaRE) guide to SQuaRE". In: *Systems Engineering* 41 (2005).

[68] International Organization for Standardization. *Software Engineering–Product Quality: Quality model*. Vol. 1. ISO/IEC, 2001.

[69] I. S. O. Iso. "IEC25010: 2011 Systems and software engineering–Systems and software Quality Requirements and Evaluation (SQuaRE)–System and software quality models". In: *International Organization for Standardization* (2011), p. 34.

[70] IEC ISO. "IEEE, Systems and Software Engineering–Vocabulary". In: *IEEE computer society, Piscataway, NJ* (2010).

[71] ISO ISO and TR IEC. “19759: Guide to the Software Engineering Body of Knowledge (SWEBOK)”. In: *International Organization for Standardization, Geneva, Switzerland* (2005).

[72] May ISO. *Systems and Software Engineering–Architecture Description*. Tech. rep. ISO/IEC/IEEE 42010, 2011.

[73] Pankaj Jalote. *An Integrated Approach to Software Engineering*. en. Springer Science & Business Media, 1997. ISBN: 978-0-387-94899-7.

[74] Jim Johnson. “ROI, It’s Your Job!” In: *Published Keynote Third International Conference on Extreme Programming*. Vol. 4. 2002, p. 48.

[75] Paul C. Jorgensen. *Software Testing: A Craftsman’s Approach, Fourth Edition*. en. CRC Press, Apr. 2016. ISBN: 978-1-4987-8578-5.

[76] Elmar Juergens, Florian Deissenboeck, and Benjamin Hummel. “Code similarities beyond copy & paste”. In: *Software Maintenance and Reengineering (CSMR), 2010 14th European Conference on*. IEEE, 2010, pp. 78–87. (Visited on 2017-01-03).

[77] Stephen H. Kan. *Metrics and Models in Software Quality Engineering*. en. Addison-Wesley Professional, 2003. ISBN: 978-0-201-72915-3.

[78] P. K. Kapur et al. *Software Reliability Assessment with OR Applications*. en. Springer Science & Business Media, May 2013. ISBN: 978-0-85729-204-9.

[79] Richard M. Karp and Michael O. Rabin. “Efficient randomized pattern-matching algorithms”. In: *IBM Journal of Research and Development* 31.2 (1987), pp. 249–260. (Visited on 2017-03-06).

[80] Wouter de Kort. *DevOps on the Microsoft Stack*. en. Apress, Apr. 2016. ISBN: 978-1-4842-1446-6.

[81] Jens Krinke. “Identifying similar code with program dependence graphs”. In: *Reverse Engineering, 2001. Proceedings. Eighth Working Conference on*. IEEE, 2001, pp. 301–309. (Visited on 2017-03-05).

[82] Klaus Krogmann. *Reconstruction of Software Component Architectures and Behaviour Models Using Static and Dynamic Analysis*. en. KIT Scientific Publishing, 2012. ISBN: 978-3-86644-804-9.

[83] Tobias Kuipers and Joost Visser. “Maintainability index revisited–position paper”. In: *Special session on system quality and maintainability (SQM 2007) of the 11th European conference on software maintenance and reengineering (CSMR 2007)*. Citeseer, 2007. (Visited on 2017-03-29).

[84] Anuradha Lakshminarayana and Timothy S. Newman. “Principal component analysis of Lack of Cohesion in Methods (LCOM) metrics”. In: *Technical Report TRUAH-CS-1999-01* (1999). (Visited on 2017-01-24).

[85] Ralf Lämmel, Joost Visser, and João Saraiva. *Generative and Transformational Techniques in Software Engineering II: International Summer School, GTTSE 2007, Braga, Portugal, July 2-7. 2007, Revised Papers*. en. Springer, Oct. 2008. ISBN: 978-3-540-88643-3.

[86] Rikard Land. "Measurements of software maintainability". In: *Proceedings of the 4th ARTES Graduate Student Conference*. 2002. (Visited on 2016-11-29).

[87] Michele Lanza and Radu Marinescu. *Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems*. en. Springer Science & Business Media, May 2007. ISBN: 978-3-540-39538-6.

[88] Philip A. Laplante. *What Every Engineer Should Know about Software Engineering*. en. CRC Press, Apr. 2007. ISBN: 978-1-4200-0674-2.

[89] Y.-S. Lee, B.-S. Liang, and F.-J. Wang. "Some complexity metrics for object-oriented programs based on information flow". In: *CompEuro'93.'Computers in Design, Manufacturing, and Production', Proceedings.* IEEE, 1993, pp. 302–310. (Visited on 2017-03-30).

[90] Meir M. Lehman. "Programs, life cycles, and laws of software evolution". In: *Proceedings of the IEEE* 68.9 (1980), pp. 1060–1076. (Visited on 2017-01-03).

[91] Jean-Louis Letouzey and Thierry Coq. "The sqale analysis model: An analysis model compliant with the representation condition for assessing the quality of software source code". In: *Advances in System Testing and Validation Lifecycle (VALID), 2010 Second International Conference on*. IEEE, 2010, pp. 43–48. (Visited on 2017-02-07).

[92] Nicole Lévy, Francisca Losavio, and Yann Pollet. "Architecture and Quality of Software Systems". In: *Software Architecture 2* (2014), pp. 133–170. (Visited on 2016-12-30).

[93] Sylviane Levy, Fernando Gamboa, et al. "Quality requirements for multimedia interactive informative systems". In: *Journal of Software Engineering and Applications* 6.08 (2013), p. 416. (Visited on 2016-12-30).

[94] Wei Li and Sallie Henry. "Object-oriented metrics that predict maintainability". In: *Journal of systems and software* 23.2 (1993), pp. 111–122. (Visited on 2017-04-21).

[95] Stanley B. Lippman, Josée Lajoie, and Barbara E. Moo. *C++ Primer*. en. Addison-Wesley, Feb. 2005. ISBN: 978-0-672-33404-7.

[96] Aldo Liso. "Software maintainability metrics model: an improvement in the Coleman-Oman model". In: *Crosstalk* (2001), pp. 15–17. (Visited on 2016-11-30).

[97] Yashwant K. Malaiya et al. "Software reliability growth with test coverage". In: *IEEE Transactions on Reliability* 51.4 (2002), pp. 420–426. (Visited on 2017-03-05).

[98] Fowler Martin. *CodeSmell*. Sept. 2006. URL: https://martinfowler.com/bliki/CodeSmell.html (visited on 2017-04-24).

[99] Jeff Martin. *Visual Studio 2015 Cookbook*. en. Packt Publishing Ltd, Aug. 2016. ISBN: 978-1-78588-593-8.

[100] Robert Martin. “OO design quality metrics”. In: *An analysis of dependencies* 12 (1994), pp. 151–170. (Visited on 2017-02-13).

[101] Robert C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. en. Pearson Education, Aug. 2008. ISBN: 978-0-13-608325-2.

[102] Robert C. Martin and Micah Martin. *Agile Principles, Patterns, and Practices in C#*. en. Pearson Education, July 2006. ISBN: 978-0-13-279714-6.

[103] Thomas J. McCabe. “A complexity measure”. In: *IEEE Transactions on software Engineering* 4 (1976), pp. 308–320. (Visited on 2017-02-11).

[104] Jim A. McCall, Paul K. Richards, and Gene F. Walters. *Factors in Software Quality. Volume-III. Preliminary Handbook on Software Quality for an Acquisiton Manager*. Tech. rep. DTIC Document, 1977. (Visited on 2017-01-31).

[105] Steve McConnell. *Code Complete*. en. Pearson Education, June 2004. ISBN: 978-0-7356-3697-2.

[106] John A. McDermid. *Software Engineer's Reference Book*. en. Elsevier, Oct. 2013. ISBN: 978-1-4831-0508-6.

[107] Tom Mens, Alexander Serebrenik, and Anthony Cleve. *Evolving Software Systems*. en. Springer Science & Business Media, Jan. 2014. ISBN: 978-3-642-45398-4.

[108] Microsoft. *Code Metrics Values*. URL: https://msdn.microsoft.com/en-us/library/bb385914.aspx (visited on 2017-02-12).

[109] Ivan Mistrik et al. *Software Quality Assurance: In Large Scale and Complex Software-intensive Systems*. en. Morgan Kaufmann, Oct. 2015. ISBN: 978-0-12-802541-3.

[110] Jamie L. Mitchell and Rex Black. *Advanced Software Testing - Vol. 3, 2nd Edition: Guide to the ISTQB Advanced Certification as an Advanced Technical Test Analyst*. en. Rocky Nook, Inc., Mar. 2015. ISBN: 978-1-4571-8910-4.

[111] Glenford J. Myers, Corey Sandler, and Tom Badgett. *The Art of Software Testing*. en. John Wiley & Sons, Sept. 2011. ISBN: 978-1-118-13315-6.

[112] Christian Nagel. *Professional C# 6 and .NET Core 1.0*. en. John Wiley & Sons, Apr. 2016. ISBN: 978-1-119-09663-4.

[113] *Nancy Blog - Working full-time with getting Nancy running on CoreCLR*. URL: http://blog.nancyfx.org/working-full-time-on-getting-nancy-running-on-coreclr/ (visited on 2017-04-29).

[114] *NancyFx/Nancy*. URL: https://github.com/NancyFx/Nancy (visited on 2017-04-29).

[115] Ali Bou Nassif, Luiz Fernando Capretz, and Danny Ho. “Software estimation in the early stages of the software life cycle”. In: *International conference on emerging trends in computer science, communication and information technology*. 2010, pp. 5–13. (Visited on 2017-02-18).

[116] J. R. Nawrocki. *Balancing Agility and Formalism in Software Engineering: Second IFIP TC 2 Central and East European Conference on Software Engineering Techniques, CEE-SET 2007, Poznan, Poland, October 10-12, 2007, Revised Selected Papers*. en. Springer Science & Business Media, Aug. 2008. ISBN: 978-3-540-85278-0.

[117] NDepend. *NDepend Code Metrics Definitions*. URL: http://www.ndepend.com/docs/code-metrics (visited on 2017-02-13).

[118] Vu Nguyen et al. "A SLOC counting standard". In: *COCOMO II Forum*. Vol. 2007. 2007. (Visited on 2017-02-18).

[119] Ariadi Nugroho, Joost Visser, and Tobias Kuipers. "An empirical model of technical debt and interest". In: *Proceedings of the 2nd Workshop on Managing Technical Debt*. ACM, 2011, pp. 1–8. (Visited on 2017-04-10).

[120] Falk Martin Oberscheven. "Software Quality Assessment in an Agile Environment". PhD thesis. Master's thesis. Radboud University Nijmegen, 2013. (Visited on 2016-12-30).

[121] Mathias Olausson and Jakob Ehn. *Continuous Delivery with Visual Studio ALM 2015*. en. Apress, Nov. 2015. ISBN: 978-1-4842-1272-1.

[122] Paul Oman and Jack Hagemeister. "Metrics for assessing a software system's maintainability". In: *Software Maintenance, 1992. Proceerdings., Conference on*. IEEE, 1992, pp. 337–344. ISBN: 0-8186-2980-0.

[123] Sergey Orlov and Andrei Vishnyakov. "Software-engineering measurement for logistics and transport systems". In: *Transport and Telecommunication* 12.3 (2011), pp. 41–53. (Visited on 2017-04-03).

[124] Maryoly Ortega, María Pérez, and Teresita Rojas. "Construction of a systemic quality model for evaluating a software product". In: *Software Quality Journal* 11.3 (2003), pp. 219–242. (Visited on 2017-02-01).

[125] Roy Osherove. *The Art of Unit Testing*. de. MITP-Verlags GmbH & Co. KG, Feb. 2015. ISBN: 978-3-8266-8722-8.

[126] Linda M. Ott and James M. Bieman. "Program slices as an abstraction for cohesion measurement". In: *Information and Software Technology* 40.11 (1998), pp. 691–699. (Visited on 2017-01-02).

[127] Linda Ott et al. "Developing measures of class cohesion for object-oriented software". In: *Proc. Annual Oregon Workshop on Software Merics (AOWSM'95)*. Vol. 11. Citeseer, 1995. (Visited on 2017-01-02).

[128] Alan Page, Ken Johnston, and Bj Rollison. *How We Test Software at Microsoft*. en. Microsoft Press, Dec. 2008. ISBN: 978-0-7356-3831-0.

[129] Nikhil Pathania. *Learning Continuous Integration with Jenkins*. en. Packt Publishing Ltd, May 2016. ISBN: 978-1-78528-503-5.

[130] Troy Pearse and Paul Oman. “Maintainability measurements on industrial source code maintenance activities”. In: *Software Maintenance, 1995. Proceedings., International Conference on*. IEEE, 1995, pp. 295–303. (Visited on 2016-11-29).

[131] Jeffrey S. Poulin. “Measuring software reusability”. In: *Software Reuse: Advances in Software Reusability, 1994. Proceedings., Third International Conference on*. IEEE, 1994, pp. 126–138. (Visited on 2017-03-30).

[132] Ruben Prieto-Diaz and Peter Freeman. “Classifying software for reusability”. In: *IEEE software* 4.1 (1987), p. 6. (Visited on 2017-03-30).

[133] Rafa E. Al-Qutaish. “Quality models in software engineering literature: an analytical and comparative study”. In: *Journal of American Science* 6.3 (2010), pp. 166–175. (Visited on 2017-04-09).

[134] Jane Radatz, Anne Geraci, and Freny Katki. “IEEE Standard Glossary of Software Engineering Terminology”. In: *IEEE Std 610.12-1990* (Dec. 1990), pp. 1–84.

[135] Mehwish Riaz, Emilia Mendes, and Ewan Tempero. “A systematic review of software maintainability prediction and metrics”. In: *Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement*. IEEE Computer Society, 2009, pp. 367–377. (Visited on 2017-04-24).

[136] Linda H. Rosenberg. “Applying and interpreting object oriented metrics”. In: (1998). (Visited on 2017-03-30).

[137] Chanchal K. Roy, James R. Cordy, and Rainer Koschke. “Comparison and evaluation of code clone detection techniques and tools: A qualitative approach”. In: *Science of Computer Programming* 74.7 (2009), pp. 470–495. (Visited on 2017-01-03).

[138] Nick Rozanski and Eóin Woods. *Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives*. en. Addison-Wesley, Oct. 2011. ISBN: 978-0-13-290612-8.

[139] Richard W. Selby. “Quantitative studies of software reuse”. In: *Software reusability*. ACM, 1989, pp. 213–233. (Visited on 2017-03-30).

[140] Raed Shatnawi. “A quantitative investigation of the acceptable risk levels of object-oriented metrics in open-source systems”. In: *IEEE Transactions on software engineering* 36.2 (2010), pp. 216–225. (Visited on 2017-02-12).

[141] N. R. Shetty, N. H. Prasad, and N. Nalini. *Emerging Research in Computing, Information, Communication and Applications: ERCICA 2015*. en. Springer, Aug. 2015. ISBN: 978-81-322-2553-9.

[142] Miguel-Angel Sicilia and Daniel Rodríguez. “Exploring Cohesion, Flexibility, Communication Overhead and Distribution for Web Services Interfaces in Computational Science”. In: *International Conference on Computational Science*. Springer, 2008, pp. 368–375. (Visited on 2017-01-24).

[143] *SIG Maintainability Model Plugin - SonarQube-4.5 - Doc SonarQube*. URL: https://docs.sonarqube.org/display/SONARQUBE45/SIG+Maintainability+Model+Plugin (visited on 2017-04-29).

[144] István Siket, Arpad Beszedes, and John Taylor. “Differences in the Definition and Calculation of the LOC Metric in Free Tools”. In: *2014* (). (Visited on 2017-02-18).

[145] Dag IK Sjøberg et al. “Quantifying the effect of code smells on maintenance effort”. In: *IEEE Transactions on Software Engineering* 39.8 (2013), pp. 1144–1156. (Visited on 2017-04-24).

[146] Alessandro Del Sole. *Visual Basic 2015 Unleashed*. en. Sams Publishing, July 2015. ISBN: 978-0-13-419670-1.

[147] SonarQube. *Metric Definitions - SonarQube Documentation - SonarQube*. URL: https://docs.sonarqube.org/display/SONAR/Metric+Definitions (visited on 2017-01-31).

[148] Diomidis Spinellis. *Code Quality: The Open Source Perspective*. en. Adobe Press, Apr. 2006. ISBN: 978-0-7686-8512-1.

[149] Girish Suryanarayana, Ganesh Samarthyam, and Tushar Sharma. *Refactoring for Software Design Smells: Managing Technical Debt*. en. Morgan Kaufmann, Nov. 2014. ISBN: 978-0-12-801646-6.

[150] Clemens Szyperski, Dominik Gruntz, and Stephan Murer. *Component Software: Beyond Object-oriented Programming*. en. Pearson Education, 2002. ISBN: 978-0-201-74572-6.

[151] Squoring Technologies. *Agile Ranking*. URL: http://www.agileranking.com/ (visited on 2017-04-30).

[152] Ricardo Terra et al. “Qualitas.class Corpus: A Compiled Version of the Qualitas Corpus”. In: *Software Engineering Notes* 38.5 (2013), pp. 1–4.

[153] Michele Tufano et al. “When and why your code starts to smell bad”. In: *Proceedings of the 37th International Conference on Software Engineering-Volume 1*. IEEE Press, 2015, pp. 403–414. (Visited on 2017-04-24).

[154] Tugberk Ugurlu and Alexander Zeitler. *Pro ASP.NET Web API: HTTP Web Services in ASP.NET*. en. Apress, Jan. 2013. ISBN: 978-1-4302-4725-8.

[155] Ali Uurlu, Alexander Zeitler, and Ali Kheyrollahi. *Pro ASP.NET Web API: HTTP Web Services in ASP.NET*. en. Apress, Nov. 2013. ISBN: 978-1-4302-4726-5.

[156] *Virtual Machinery - Sidebar 4 - MI and MINC - Maintainability Index*. URL: http://www.virtualmachinery.com/sidebar4.htm (visited on 2017-04-29).

[157] Joost Visser et al. *Building Maintainable Software, Java Edition: Ten Guidelines for Future-Proof Code*. en. "O’Reilly Media, Inc.", Jan. 2016. ISBN: 978-1-4919-5351-8.

[158] Gwendolyn H. Walton, Jesse H. Poore, and Carmen J. Trammell. “Statistical testing of software based on a usage model”. In: *Software: Practice and Experience* 25.1 (1995), pp. 97–108. (Visited on 2017-05-02).

[159] Arthur Henry Watson, Dolores R. Wallace, and Thomas J. McCabe. *Structured testing: A testing methodology using the cyclomatic complexity metric*. Vol. 500. 235. US Department of Commerce, Technology Administration, National Institute of Standards and Technology, 1996. (Visited on 2017-04-03).

[160] Kurt D. Welker. “The software maintainability index revisited”. In: *CrossTalk* 14 (2001), pp. 18–21. (Visited on 2017-04-08).

[161] Niklaus Wirth. “The module: A system structuring facility in high-level programming languages”. In: *Language Design and Programming Methodology*. Springer, 1980, pp. 1–24. (Visited on 2017-01-03).

[162] Eberhard Wolff. *A Practical Guide to Continuous Delivery*. en. Addison-Wesley Professional, Feb. 2017. ISBN: 978-0-13-469154-1.

[163] Aiko Yamashita and Leon Moonen. “Do code smells reflect important maintainability aspects?” In: *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*. IEEE, 2012, pp. 306–315. (Visited on 2017-03-05).

[164] Hongji Yang and Dr Martin Ward. *Successful Evolution of Software Systems*. en. Artech House, 2003. ISBN: 978-1-58053-588-5.

[165] Shlomo Yitzhaki. “On an extension of the Gini inequality index”. In: *International economic review* (1983), pp. 617–628. (Visited on 2017-03-04).

[166] Shlomo Yitzhaki. “Relative deprivation and the Gini coefficient”. In: *The quarterly journal of economics* (1979), pp. 321–324. (Visited on 2017-03-04).

[167] Edward Yourdon and Larry L. Constantine. *Structured design: Fundamentals of a discipline of computer program and systems design*. Prentice-Hall, Inc., 1979. ISBN: 978-0-13-854471-3. (Visited on 2017-01-02).

[168] Yang Yuan and Yao Guo. “CMCD: Count matrix based code clone detection”. In: *Software Engineering Conference (APSEC), 2011 18th Asia Pacific*. IEEE, 2011, pp. 250–257. (Visited on 2017-03-05).

[169] Jasni Mohamad Zain, Wan Maseri Wan Mohd, and Eyas El-Qawasmeh. *Software Engineering and Computer Systems, Part I: Second International Conference, ICSECS 2011, Kuantan, Malaysia, June 27-29, 2011. Proceedings*. en. Springer, June 2011. ISBN: 978-3-642-22170-5.

List of Figures

List of Tables

List of Code

List of Abbreviations

CBB	Codebase Balance
CBO	Coupling between Object Classes
CBS	Codebase Size
CC	Cyclomatic Complexity
CD	Continuous Delivery
CI	Continuous Integration
CoC	Cost of Change
CPD	Copy Paste Detection
CQLinq	Code Query for Language Integrated Query
CRUD	Create, Read, Update, Delete
DIT	Depth of Inheritance Tree
DMS	Distance from Main Sequence
IOC	Inversion of Control
JSON	JavaScript Object Notation
LCOM	Lack of Cohesion of Methods
LINQ	Language Integrated Query
LLOC	Logical Line of Code
MI	Maintainability Index
NDMR	NDepend Maintainability Rating
NOC	Number of Children
RC	Relational Cohesion
SLOC	Source Line of Code

SQALE	Software Quality Assessment based on Lifecycle Expectations
SQMR	SonarQube Maintainability Rating
SQUARE	Software Engineering – Software Product Quality Requirements and Evaluation
TC	Test Coverage
UI	User Interface
US	Unit Size
VS2017	Visual Studio Enterprise 2017
VSMI	Visual Studio flavoured Maintainability Index
WCF	Windows Communication Framework