Metric-based Evaluation of Implemented Software Architectures

PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Technische Universiteit Delft,
op gezag van de Rector Magnificus Prof. Ir. K.Ch.A.M. Luyben
voorzitter van het College voor Promoties,
in het openbaar te verdedigen op vrijdag 28 juni 2013 om 15.00 uur

door
Eric Matteas BOUWERS,
Master of Science in Computer Science,
geboren te Rijswijk.

Dit proefschrift is goedgekeurd door de promotoren:

Prof. dr. A. van Deursen
Prof. dr. ir. J.M.W. Visser

Samenstelling promotiecommissie:

Rector Magnificus	voorzitter
Prof. dr. A. van Deursen	Technische Universiteit Delft, promotor
Prof. dr. ir. J.M.W. Visser	Radboud Universiteit Nijmegen, promotor
Prof. dr. ir. G.J. Houben	Technische Universiteit Delft
Prof. dr. P. B. Kruchten, P.Eng.	University of British Columbia, Canada
Prof. dr. ir. R. L. Lagendijk	Technische Universiteit Delft
Prof. dr. A. Zeller	Universität des Saarlandes, Duitsland
Dr. P. Lago	Vrije Universiteit Amsterdam

ISBN 9789088916380

Typeset by the author with the LaTeX Documentation System.
Printed by Proefschriftmaken.nl || Uitgeverij BOXPress.
Cover design by Proefschriftmaken.nl.

Author email: e.bouwers@sig.eu

Acknowledgments

In December 2007 I had my first evaluation review at the Software Improvement Group with Per John and Brigitte van der Vliet. During this review they asked me whether I wanted to pursue a PhD, a possibility I did not realize I had until then. I don't think I would have even considered going back into research without them asking me for it. The fact that you can now read this thesis shows that I am very happy that they did.

The next 200 pages provide an overview of the research I have conducted in the past four and a half years. This research has not only lead to the results presented in this thesis, but also allowed me to get into contact with many interesting researchers, travel to locations I would not think of visiting, and made me aware of aspects of myself I did not know I had. All of this, and much more, would certainly not have been possible without the help and support of many different people.

To start, I would like to thank Arie and Joost for their willingness to supervise me. The results in this thesis would not be presentable without the questions, remarks and advice you both gave me during all hours of the day. I greatly enjoyed our discussions and hope to have more of them in the future.

Then I want to thank all of my current and former colleagues at the Software Improvement Group for their help. Not only have you helped me by planning around my research time, you have also assisted me by sharing your ideas, participating in experiments, and by always offering your feedback on the results. Thank you for offering me such a rich and inspiring environment.

Similarly, I would like to thank the colleagues of the Software Engineering Research Group at Delft University of Technology. Even though I was there only once a week (at best), you still made me feel at home and did not hesitate to help me in both Delft and abroad.

A special thanks goes out to all of my co-authors for their willingness to work together on conducting joint research and writing down the results. I have learned new things from all of you, which are certainly going to be useful in the remainder of my career.

Lastly, I want to thank my friends and family for listening to me and supporting me during all these years. Thank you for distracting me by going climbing, playing games, or sharing dinner with me. And mom and dad in particular, thank you for always supporting me and for allowing me to do what I wanted to do.

But above all, thank you Karin for sticking with me and helping me in every way possible. It is unbelievable how many great gifts you have given me! I deeply love you and Suus and look forward to spend the rest of my life making you happy.

Eric Bouwers
Maarssen, April 2013

Contents

1 Introduction 1

1.1 Software Architecture: What to evaluate? . . . 2

1.2 Software Architecture: How to evaluate? . . . 4

1.3 Research Context . . . 7

1.4 Research Method . . . 8

1.5 Research Approach . . . 9

1.6 Origin of Chapters . . . 11

2 Criteria for the Evaluation of Implemented Architectures 15

2.1 Introduction . . . 15

2.2 Software Risk Assessments . . . 17

2.3 Architectural System Properties . . . 19

2.4 Problem Statement . . . 20

2.5 Empirical Study . . . 21

2.6 Answers to research questions . . . 27

2.7 Discussion . . . 29

2.8 Related Work . . . 31

2.9 Conclusion . . . 32

3 A Cognitive Model for Software Architecture Complexity 33

3.1 Introduction . . . 33

3.2 Scope and Contribution . . . 34

3.3 Background . . . 35

3.4 Overview of SACM . . . 38

3.5 Criteria of SACM . . . 42

3.6 Discussion . . . 46

3.7 Conclusion 50

4 A Lightweight Sanity Check for Implemented Architectures 51
4.1 Introduction 51
4.2 Background 52
4.3 LiSCIA 53
4.4 Discussion 59
4.5 Related Work 60
4.6 Conclusion 61

5 Getting what you measure: four common pitfalls in using software metrics 63
5.1 Software metrics steer people 64
5.2 What does the metric mean? 65
5.3 How many metrics do you need? 69
5.4 Conclusion 70

6 Quantifying the Analyzability of Software Architectures 73
6.1 Introduction 73
6.2 Problem statement 75
6.3 Requirements 76
6.4 Related Work 77
6.5 Counting components 78
6.6 Metric definition 80
6.7 Evaluation Design 84
6.8 Quantitative evaluation of metric performance 85
6.9 Case study 88
6.10 Discussion 91
6.11 Threats to Validity 93
6.12 Conclusions 94

7 Dependency Profiles for Software Architecture Evaluations 95
7.1 Introduction 95
7.2 Background 96
7.3 Dependency Profiles 97
7.4 Preliminary Observations 100
7.5 Discussion 102
7.6 Evaluation Design 102
7.7 Conclusions 103

8 Quantifying the Encapsulation of Implemented Software Architectures 105
8.1 Introduction . . . 106
8.2 Problem Statement . . . 107
8.3 Metrics for Encapsulation . . . 107
8.4 Experiment Design . . . 113
8.5 Experiment Implementation . . . 119
8.6 Experiment Results . . . 123
8.7 Discussion . . . 128
8.8 Threats to Validity . . . 130
8.9 Related Work . . . 132
8.10 Conclusion . . . 133
8.11 Architecture Metrics Overview . . . 134

9 Evaluating Usefulness of Software Metrics - an Industrial Experience Report 137
9.1 Introduction . . . 138
9.2 Evaluation Goal . . . 139
9.3 Evaluation Method . . . 140
9.4 Evaluation Setting . . . 140
9.5 Data Gathering . . . 144
9.6 Observation Findings . . . 145
9.7 Interview Findings . . . 150
9.8 Discussion of Findings . . . 154
9.9 Reflections on Evaluation Methodology . . . 155
9.10 Related Work . . . 156
9.11 Conclusion . . . 157

10 Conclusion 159
10.1 Contributions . . . 159
10.2 Answer to Research Questions . . . 160
10.3 Impact on Practice . . . 162
10.4 Impact on Research . . . 163
10.5 Future work . . . 164

Bibliography 166

Summary 176

Samenvatting 180

Curriculum Vitae 184

Appendix A LiSCIA **187**

A.1 Goal 187

A.2 Start-up Phase 188

A.3 Review Phase 190

A.4 Actions and Guidelines 193

CHAPTER 1

Introduction

Between waking up in the morning and reading the first e-mail at work the majority of people have already interacted with over a dozen software systems. All aspects of a daily commute, from waking up in the morning by the sound of the radio to actually arriving at work using a car or public transportation, is controlled by software systems.

As with all man-made objects, the functionality of a software system is bounded by the principles of its construction. Just as a car typically consists of wheels, a body and an engine which enables ground-transportation (but not flying), a software system designed for an alarm-clock will not be able to track trains. Within software engineering, this framework of principles is typically referred to as the *software architecture* of a system.

Software architecture is loosely defined as “the organizational structure of a software system including components, connections, constraints, and rationale” (Kogut and Clements, 1994). All software systems are inherently constraint by a software architecture which may be the result of a deliberate process of balancing stakeholder requirements, may have organically grown by accumulating choices made by individual developers, or anything in between.

Although the process of creating the architecture might deviate from project to project, the importance of the architecture within the system is paramount, as stated by Clements et al. (2002): *“Architectures allow or preclude nearly all of the system’s quality attributes”*. In other words, without a proper architecture the quality of system properties cannot be ensured or can be highly expensive, or even impossible, to implement.

For example, a software system which is designed and constructed as an iOS app cannot be easily ported to a Windows-based operating system because of the inherent use of operating system specific procedures. Similarly, the architectural choice for building a web-based software system has the consequence that some functionality, such as using operating system specific user interface elements or the direct manipulation of files on the client, is not possible, or at least very challenging and costly to implement.

Whether these limitations are problematic is context-dependent. The software that controls the break-installation of a car does not necessarily need a graphical user-interface, while the software system within a navigation device cannot function without one. Additionally, for a system that is only used for a one-time conversion of data the maintainability of the code is probably less important than the speed or correctness of the conversion.

In order to determine whether the limitations imposed by the architecture are problematic it is crucial to evaluate the architecture of a software system in light of the current context and future plans with the system. Within this thesis we take a closer look into both the “what” and “how” of architecture evaluations.

1.1 Software Architecture: What to evaluate?

To assist experts in the evaluation of a software architecture various methodologies have been developed (see for example Babar et al. (2004) and Dobrica and Niemelä (2002) for two different overviews). These methodologies have been divided into two different types of evaluations, *early* architecture evaluations which focus on a designed architecture and so-called *late* architecture evaluations focussing on the architecture already implemented in a software system (Dobrica and Niemelä, 2002).

Ideally, the designed architecture of a system is evaluated before the implementation of the system begins, and also when the context or future plans of the system change significantly. When the implemented architecture strictly follows the design, the quality attributes ensured by the designed architecture are also ensured by the implementation, limiting the need for more frequent evaluations of the implemented architecture.

However, in practice we often encounter systems for which it is not possible, nor effective, to evaluate the designed architecture. In some cases there never was a designed architecture, in other cases the documentation of the architecture is out-of-date, e.g., the design and the implementation of the architecture are out-of-sync. Also, we frequently encounter up-to-date documentation which is incomplete, e.g., important trade-offs are either not documented or not considered at all.

Therefore, evaluating a designed architecture in isolation, i.e., without taking into account the implemented architecture, leads to an incomplete overview of the

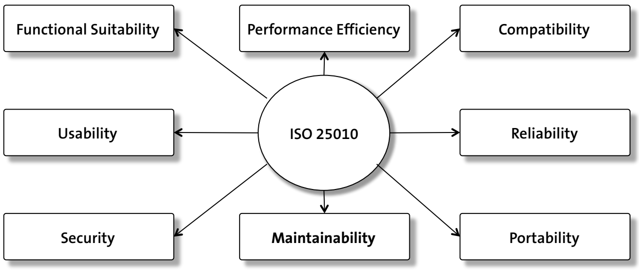


Figure 1.1: *The eight quality characteristics of software quality as defined by the International Organization for Standardization (2011)*

strengths and weaknesses of a software system. Using such an incomplete view of the architecture as a basis for decision-making can have severe consequences, potentially including the loss of revenue or reduced credibility of the company. Because of these observations the focus of this thesis is on the evaluation of existing systems based on their *implemented* architecture.

To evaluate an implemented architecture many different criteria can be considered. Based on the evaluation of designed architecture, criteria such as the component-structure and the dependencies between the components of a system are likely candidates to be evaluated. However, there is currently no overview of all criteria that should be evaluated (or can safely be ignored).

A first step towards such a set of criteria is to choose a quality characteristic of a system which is in need of evaluation. According to the International Organization for Standardization (2011), the quality of a software system can be analyzed using eight different quality characteristics, see Figure 1.1 for an overview.

Each quality characteristic is decomposed into several sub-characteristics. For example, *Maintainability* is decomposed into: Analyzability, Modifiability, Testability, Modularity and Reusability. The standard provides a definition for each (sub-)characteristic, but does not provide an overview of the *criteria* which should be used to evaluate the sub-characteristics. This makes the identification of these criteria an open research topic.

The choice of a particular quality characteristic as defined in the ISO/IEC 25010 will affect the criteria. For example, choosing *Performance* will most likely result in criteria related to “response time” and “memory utilization”. In contrast, a choice for *Security* will likely yield criteria such as “secure communication” and “encryption strength”.

In this research, we focus on the *Maintainability* characteristic of software quality, which is defined as:

> Degree of effectiveness and efficiency with which a product or system can be modified by the intended maintainers (International Organization for Standardization, 2011).

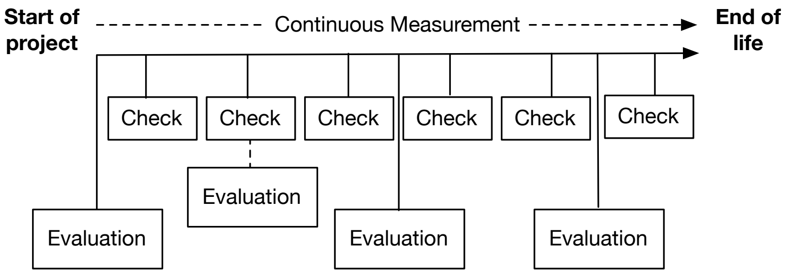


Figure 1.2: *Overview of activities ideally conducted during a software project*

Based on this definition we reason that this quality characteristic is a prerequisite to achieve any of the other quality characteristics efficiently. This leads us to the first research question:

RQ1: Which criteria should be taken into account during the evaluation of the maintainability of an implemented architecture?

1.2 Software Architecture: How to evaluate?

In an ideal situation a project is under *Continuous Software Architecture Evaluation*, i.e., there is a constant attention towards the state of the software architecture of the system using different types of activities as shown in Figure 1.2.

The evaluation of a designed architecture is done at the beginning of a project to ensure that the design fits the current business needs. During the implementation and maintenance phase this evaluation should be repeated at every point in time in which the current design is not able to meet the (future) needs from the business. If this situation is detected the evaluation team, the architects of the system, and representatives from important stakeholders should align, evaluate the current design and define actions to make any adjustments which are necessary to either the design or the implementation.

The evaluation of a designed architecture can take several days and involves a relatively large group of people. Therefore it is not cost-efficient to perform such an evaluation on a regular basis, for example every three months. Instead, it would be better to perform such an evaluation only when it is needed, e.g., only when the business-context or important characteristics of the implemented architecture changes.

To detect significant changes in the important characteristics of an implemented architecture we envision the use of a check-list (Gawande, 2009). Such a check-list can be performed by a *quality evaluator*, a role which can either be full-filled by a team-member or by an external party. By filling in a check-list on a regular

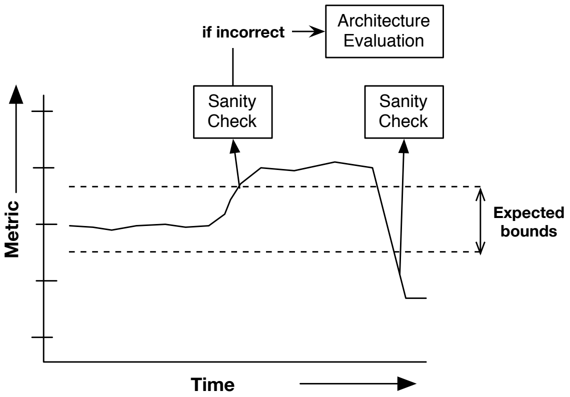


Figure 1.3: *Triggers within a continuous software architecture evaluation setting*

basis significant changes in the implemented architecture can be detected and a full evaluation of the designed architecture can be scheduled if needed.

Better yet, because the implemented architecture of a system is embedded in the software system itself we can use software metrics to continuously monitor important aspects of the implemented architecture. Basic metrics, for example the number of components, are straight-forward to calculate after each change and can be used as a trigger-mechanism by the quality evaluator.

Whenever a metric deviates beyond its expected bounds it is the responsibility of the evaluator to determine the root-cause of the change by interpreting the (change in) value of the metric. If this root-cause indicates a problem the check-list can be used to determine other potential problems, which are then discussed with the development team and the architects to explore alternative solution strategies. Should this fail a more full-scale architecture evaluation could be warranted.

Consider the following situation, based on our experience, as an example. During the development of a new system a component called “XX-connector” is introduced which implements a connection to the XX-system, but also includes logic for when the XX-system is not available. This increased number of components triggers the quality evaluator to check the current implemented architecture which, amongst others, consists of a component called “YY-connector” which also implements both the connection and fail-over logic. From a previous discussion with the architects the quality evaluator remembers that the system needs to communicate with a wide range of software systems.

If no action is taken this could result in a large number of connector-components, each of which implements its own fail-over mechanism. To prevent this situation, the quality evaluator initiates a discussion with the development team to ensure that the connection to external systems, including fail-over mechanisms, are implemented in a consistent manner.

Figure 1.3 provides an illustration of the trigger-process as described above. According to our own experience, this type of repeated evaluation of either the designed or implemented architecture of a system is often not embedded within development projects. The research literature provides some hints as to why this is the case.

1.2.1 Current State of Art: Architecture Evaluations

The results of a survey conducted by Babar and Gorton (2009) shows that the adoption of any form of (structural) architecture evaluations within industry is low. One of the reasons given by Babar et al. for this low level of adoption is the lack of process and tool-support for companies that want to start performing architecture evaluations. Following from these conclusions we define our second research question:

> RQ2: What support can we define to make the process of regularly checking an implemented architecture easier for a quality evaluator?

1.2.2 Current State of Art: Architecture Metrics

As mentioned before, the concept of *Maintainability* is decomposed into five different sub-characteristics. One of these characteristics, *Modularity*, directly targets to the software architecture of a system. Note how the definition of this characteristic describes desirable properties of the components of a system:

> Degree to which a system or computer program is composed of discrete components such that a change to one component has minimal impact on other components (International Organization for Standardization, 2011)

According to a recent survey performed by Koziolek (2011), there exists 19 architecture-level metrics which aim to evaluate this property. However, despite their availability the repeated application of these metrics is not often seen in practice.

One reason for not regularly using architecture metrics within a software development process could be the fact that most metrics lack an empirical validation (Koziolek, 2011), e.g., it is not clear whether the available metrics indeed quantify the desired quality characteristic. This property is called “construct validity” (Kaner and Bond, 2004).

For those metrics which have been evaluated, the evaluation normally focusses on the construct validity of the specific metric while the comparison with existing metrics is hardly ever performed. Thus it is yet unclear which architecture metrics are most suitable to quantify the modularization of a software system, which leads us to our third research question:

> RQ3: Which metrics are capable of quantifying the modularization of a software system?

Apart from evaluations which cover the construct validity of a metric there exist some evaluations which focus on the mathematical (or metrological) properties of a metric. However, the *usefulness* of the metric is normally not evaluated. In other words, the question "does the use of the metric help a quality evaluator in initiating the right type of discussions?" is never considered. This leads us to our fourth and final research question:

RQ4: Are the metrics identified in RQ3 useful in practice?

1.3 Research Context

The research presented within this thesis has been conducted within the Software Improvement Group (SIG), a Dutch consultancy firm which "...translates detailed technical findings concerning software-intensive systems into actionable advice for upper management."[1]. SIG is continuously seeking to improve its way of working. To that end, SIG collaborates with the research community, and frequently publishes about the evaluation methodology used (Deursen and Kuipers, 2003; Kuipers and Visser, 2004; Heitlager et al., 2007; Baggen et al., 2010).

All research experiments have been performed within the period from October 2008 to December 2012. During his research the author was also involved in the delivery of the services offered by SIG. Prior to starting this research, the author worked at SIG as a technical consultant for a period of one year. The insights into real-world problems and the experiences with different solution strategies gained during the delivery of these services have also been an important motivator for conducting the research leading to this thesis.

Within the services of SIG, a team of consultants has the role of external quality evaluators for software systems built or maintained by the clients of SIG or their suppliers. In the initial working period, the evaluation of the architecture of a system was part of every project, but a structured approach was missing. Instead, the evaluations relied heavily on the expertise of the individual consultants, which makes it hard to deploy the services on a larger scale, increases the time to educate new personnel and limits the traceability and repeatability of the evaluations.

Based on these observations, we hypothesize that a more structured approach towards the evaluation of an implemented architecture, such as described in Section 1.2, makes it possible to lift the various limitations of expert-based evaluations. We consider the research project successful if the answers to the research questions allow such a structured approach to be implemented in practice, or if the answers show that such a structured approach is infeasible.

[1] `http://www.sig.eu`

1.4 Research Method

Even though this research project has been initiated based on a problem identified in a particular research context, the research of Babar and Gorton (2009) shows that the lack of a structured approach towards software architecture evaluation is a more general problem. This means that the answers to the research questions should also be useable by quality evaluators in general. If this is not the case, i.e., if the answers to the research questions are not valid outside our specific research context, then we consider this research unsuccessful.

The research approach we use can be defined as "industry-as-a-laboratory" (Potts, 1993). In this approach a researcher closely collaborates with industry to identify problems and create and evaluate solutions. To ensure that the answers to the research questions can still be applied outside our specific research context, we use a mix of well-established research methods, each of which is selected based on the nature of a specific research question (Creswell and Clark, 2006).

For RQ1 we extract criteria from practice taking into account guidelines from grounded theory (Adolph et al., 2011), which we then validate using interviews. In addition, the theoretical validity of these criteria is done by matching the identified criteria against theories taken from cognitive science (Hutchins, 1996).

To answer RQ2 we build upon the theoretical framework which is the result of RQ1 and combine this with our own experience in the metric-based evaluation of implemented software architectures. Since our goal here is to construct hands-on advice for practitioners we validate the interest of a more general public by targeting publication venue's which are oriented towards practitioners.

In order to answer RQ3 we quantitatively evaluate the construct validity of potential metrics using experiments and case-studies. By following the guidelines from Wohlin et al. (2000) and Yin (2009) we ensure that conclusions drawn are valid and that the results are as generic as possible.

Lastly, since the aim of RQ4 is to determine which metrics are useful we perform a qualitative evaluation by executing a large-scale empirical study, again using the guidelines from grounded theory (Adolph et al., 2011). In addition, we conduct interviews with experienced quality evaluators to determine whether the most promising metrics are indeed useful.

Whether the research method outlined above indeed results in a balanced and generally applicable answer to the research questions is discussed in Chapter 10. This chapter also discusses the impact of the results on both practice and the research community.

1.5 Research Approach

Based on the methodological considerations outlined in the previous section we define a more specific approach for each research question below.

RQ1: Which criteria should be taken into account during the evaluation of the maintainability of an implemented architecture?

The research context of SIG provides an unique opportunity to access quality evaluators and data extracted from industry systems, both in the form of raw metric-data as well as their interpretation in written reports. This opportunity is leveraged to answer RQ1 by mining the contents of evaluation reports containing an evaluation of the implemented architecture of a wide range of systems. To validate the analyses, interviews are conducted with two experienced consultants. The result of this study is a set of criteria which have been used to evaluate the implemented architecture of a software system. Details about the design and execution of this study are given in Chapter 2.

The extraction of the criteria from empirical data provides us with insight into which criteria have been used to evaluate implemented architecture, but it does not necessarily explain *why* these criteria influence the maintainability of an implemented architecture (if they have any influence at all). To define these relationships we extend the existing architecture complexity model of Lilienthal (2009) based on theories taken from cognitive science. The result of this study is the Software Architecture Complexity Model (SACM), of which a description is given in Chapter 3.

As a result, those criteria which are extracted from the empirical data and which can be explained in terms of the SACM should provide the answer to RQ1.

RQ2: What support can we define to make the process of regularly checking an implemented architecture easier for a quality evaluator?

Apart from answering RQ1 the criteria identified in Chapter 2 also serve as a basis for the definition of a Lightweight Sanity Check for Implemented Architectures (LiSCIA). By combining the specific criteria with experiences gained from structurally evaluating the maintainability of software systems, as defined by Deursen and Kuipers (2003) and Kuipers and Visser (2004), we design an easy-to-use check-list which can be executed within a day. A description of LiSCIA and its design decisions is given in Chapter 4.

Another use of our experiences in performing metric-based evaluations of the maintainability of software systems is the definition of practical guidelines. During the execution of the services of SIG we identified four major failure patterns in the use of software metrics in a project management setting. In our experience, knowing these patterns enables a manager to avoid unwanted situations, which makes it easier to reach a pre-defined goal. Chapter 5 discusses these patterns using several examples taken from practice.

As a result, both chapters provide actionable advice which enables practitioners to perform regular evaluations of an implemented architecture using metrics, thus answering RQ2.

RQ3: Which metrics are capable of quantifying the modularization of a software system?

Keeping the different pitfalls in mind we aim to identify two metrics which quantify two different aspects of the modularity of an implemented architecture. Taking into account the observation of Koziolek (2011) that most metrics lack validation, our focus will be on the empirical validation of the construct validity of existing metrics, and introduce new metrics if needed.

The definition of software architecture mentions two aspects of modularization which we want to capture in two different metrics: components and connections. Thus, one of the metrics must quantify the analyzability of a system in terms of its components, while the other metric focusses on the connections between these components in relation to the encapsulation within a system.

To identify a metric which can quantify the analyzability of a system in terms of its components we first inspect the componentization of a large set of systems. Based on this data we derive a metric called Component Balance (*CB*) which combines both the number of components as well as their (difference in) volume. By performing both a quantitative evaluation, in which we correlate the opinion of experts to the metric-values, as well as a qualitative evaluation, by performing a case-study, we evaluate the initial applicability of *CB*. Details about the design and evaluation of *CB* are given in Chapter 6.

Our proposal for a metric to quantify the dependency between components is called a Dependency Profile. Within this profile, all source-code modules of a system are categorized into one of four categories depending on their relationship with modules in different components. The initial hypothesis is that such a profile is an improved indicator of the dependencies between components as opposed to simply counting the number of dependencies. The design and initial evaluation of the Dependency Profile is given in Chapter 7.

To test the initial hypothesis we perform an empirical evaluation in which the historic encapsulation of 10 open-source software systems is correlated with the values of 12 architecture level metrics, including three categories of the Dependency Profile. The results of this study show a moderate correlation between the three categories of the Dependency Profile and the metric for historic encapsulation. For the remaining nine dependency-metrics the correlation is either small or not significant. The details of this empirical study are given in Chapter 8.

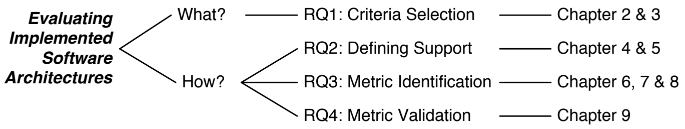


Figure 1.4: *Schematic overview of the relationship between the subject of this thesis, the research questions and the chapters*

The results of both experiments should identify those metrics which are best suited to quantify two important properties of the modularization characteristic, thus answering RQ3.

RQ4: Are the metrics identified in RQ3 useful in practice?

Based on the results as presented in Chapter 6 and Chapter 8 we move forward to a large-scale evaluation of the two metrics using a four step method. First, the metrics are embedded in the standard operating procedure of the quality evaluators within SIG. Afterwards, data about their experiences is gathered using two different methods (in two consecutive steps). Lastly, gathered data is analyzed to derive both an overview of situations in which the metrics are useful, as well as possible areas of improvement for the specific metrics. Chapter 9 provides an overview of the design and execution of this large-scale evaluation.

1.6 Origin of Chapters

Figure 1.4 shows a breakdown of the subject of this thesis, the relationship between the topic and the research questions, and the connection to the different chapters. Lastly, Chapter 10 discusses the contributions of this thesis, the answer to the research question and outlines areas of future work.

Most chapters in this thesis are based on a peer-reviewed publication, Chapter 8 and Chapter 9 have been submitted for review. Each chapter contains some redundancy in the explanation of context and related work to keep it self-contained, thus each chapter can be read in separation. The author of this thesis is the main author of all publications.

- The introduction is loosely based on two sources. The first one is a book-chapter written in Dutch called *Ontwerp versus implementatie - de kans om ze niet uiteen te laten lopen*[*], published as part of the Landelijk Architectuur Congres 2010.
  The second source is an abstract of a presentation called *Continuous Architecture Evaluation* presented at the 10th BElgian-NEtherlands software eVOLution seminar (BENEVOL 2011).

[*] Design versus implementation - the chance to keep them aligned

- Chapter 2 is based on the paper *Criteria for the Evaluation of Implemented Architectures*, which appeared in the proceedings of the 25th IEEE International Conference on Software Maintenance (ICSM 2009). This paper is referenced as (Bouwers et al., 2009).

- Chapter 3 is an extended version of the paper *A Cognitive Model for Software Architecture Complexity*, which appeared in the proceedings of the 2010 IEEE 18th International Conference on Program Comprehension (ICPC 2010). This paper is referenced as (Bouwers et al., 2010).

- Chapter 4 is based on the article *A Lightweight Sanity Check for Implemented Architectures*, which appeared in IEEE Software, Volume 27 number 4, July 2010. This article is referenced as (Bouwers and van Deursen, 2010).

- Chapter 5 is based on an article *Getting what you measure*, which appeared in Communications of the ACM, Volume 55 Issue 7, July 2012. This article is referenced as (Bouwers et al., 2012).

- Chapter 6 is based on the paper *Quantifying the Analyzability of Software Architectures*, which appeared in the proceedings of the 9th Working IEEE/IFIP Conference on Software Architecture (WICSA 2011). This paper is referenced as (Bouwers et al., 2011a).

- Chapter 7 is based on the short-paper *Quantifying the Encapsulation of Implemented Software Architectures*, which appeared in the proceedings of the 27th IEEE International Conference on Software Maintenance (ICSM 2011). This paper is referenced as (Bouwers et al., 2011c).

- Chapter 8 is based on the paper *Quantifying the Encapsulation of Implemented Software Architectures*, which has been submitted to ACM Transactions on Software Engineering and Methodology on September 6th, 2012. A technical report of this paper is available as (Bouwers et al., 2011b).

- Chapter 9 is based on the paper *Evaluating Usefulness of Software Metrics*, which appeared in the proceedings of the 35th International Conference on Software Engineering (ICSE 2013), Software Engineering in Practice (SEIP) track. This paper is referenced as (Bouwers et al., 2013).

Apart from these publications the author has been involved in the following publications which are not directly included in this thesis:

- *Detection of Seed Methods for Quantification of Feature Confinement*, which appeared in the proceedings of the 50th International Conference on Objects, Models, Components, Patterns (TOOLS Europe 2012). This paper is referenced as (Olszak et al., 2012).

- *Preparing for a Literature Survey of Software Architecture using Formal Concept Analysis*, which appeared in the proceedings of the Fifth International Workshop on Software Quality and Maintainability (SQM'11) at CSMR 2011. This paper is referenced as (Couto et al., 2011).

- *Extracting Dynamic Dependencies between Web Services Using Vector Clocks*, which appeared in the proceedings of the IEEE International Conference on Service Oriented Computing & Applications (SOCA 2011). This paper is referenced as (Romano et al., 2011).

CHAPTER 2

Criteria for the Evaluation of Implemented Architectures [*]

Abstract

Software architecture evaluation methods aim at identifying potential maintainability problems for a given architecture. Several of these methods exist, which typically prescribe the structure of the evaluation process. Often left implicit, however, are the concrete system attributes that need to be studied in order to assess the maintainability of implemented architectures.

To determine this set of attributes, we have performed an empirical study on over 40 commercial architectural evaluations conducted during the past two years as part of a systematic "Software Risk Assessment". We present this study and explain how the identified attributes can be projected on various architectural system properties, which provides an overview of criteria for the evaluation of the maintainability of implemented software architectures.

2.1 Introduction

Any active software system will need maintenance in order to keep up with new demands and changing business requirements (Lehman, 1980). From this perspective, a good software architecture is desired because, according to Clements et al. (2002); *"Architectures allow or preclude nearly all of the system's quality attributes"*. Because of this, it is not surprising that a wide range of software architecture evaluation methodologies exists (for overviews see Babar et al. (2004) and Dobrica and Niemelä (2002)) for selecting an architecture that minimizes business risks.

[*] Originally published in the proceedings of the 25th IEEE International Conference on Software Maintenance (ICSM 2009) (Bouwers et al., 2009). Terminology has been updated to fit the thesis.

Examining the review of Babar et al. (2004), we conclude that almost all of the discussed methods focus on evaluating the quality of a designed architecture, i.e., evaluating the architecture before it is implemented. In contrast, the so-called *late architecture evaluations* (Lindvall et al., 2003) are focused on assessing the quality of an *implemented architecture*. Taking a closer look at the late architectural evaluation methods we notice that they only define the structure of the evaluation in the form of roles (e.g., evaluation team, architect, stakeholders) and steps (e.g., the nine steps of the ATAM (Clements et al., 2002)). Although this structure provides a basic framework, it does not explain which properties of a system should be studied. Usually, finding out which properties to study is part of the process itself.

Fortunately, there is research available that provides examples of system properties to study, see for example Kazman and Carrière (1999) or Murphy et al. (1995). These techniques mainly focus on extracting a high-level (component) view of a system in terms of components and their dependency relations. This view is then compared with a previously designed architecture. In this light, the quality of the implemented architecture is directly coupled with the conformance to the original designed architecture.

Unfortunately, in many cases the documentation of the architecture is not available or out-of-date (Lilienthal, 2009). Also, the architecture of a system contains more than the relationships amongst the main components. Many researchers agree that one needs to inspect a system using multiple views to get a complete overview of the architecture of a system (Bass et al., 2003; Kruchten, 1995).

In order to evaluate the maintainability of a system, the Software Improvement Group (SIG) has developed the source-based Software Risk Assessments (SRA) method (Deursen and Kuipers, 2003), which it uses to assess systems on a commercial basis. Part of this method is dedicated to evaluating the implemented architecture of a software system. During the course of an SRA, a Maintainability Model (Heitlager et al., 2007) is used. This model provides an overview of several system properties to consider, including three system properties which address architectural issues from different perspectives.

Most system properties used within the Maintainability Model are assessed by auditing a single system attribute. For example, the system attribute *lines of code* is used to assess the system property *Volume*. Unfortunately, architectural system properties are often too broad to be assessed by a single system attribute. Instead, several system attributes need to be judged and combined to come to a balanced quality rating. This quality rating is currently based on expert opinion. To avoid inconsistencies in the quality ratings, systems are always assessed by multiple experts. However, it would be beneficial if the rating of an architecture's quality could be (partially) derived in a more formalized way.

To achieve this we have conducted an empirical study to reach two goals: 1) identifying the system attributes the SIG experts have used to assess these architectural system properties, and 2) finding out how the system attributes are normally projected onto the three architectural system properties. Combining the answers to these research questions leads to an overview of criteria for evaluating the maintainability of implemented architectures.

This chapter is structured as follows: we first introduce the environment in which the SIG evaluates implemented architectures in Section 2.2. After this, the architectural properties are introduced in Section 2.3. A problem statement and ensuing research questions are formulated in Section 2.4, followed by the design of our empirical study in Section 2.5. The results of the study are used to formulate answers to the research questions in Section 2.6. A discussion of the relevance of the study and threats to its validity is provided in Section 2.7, after which related work is discussed in Section 2.8. Finally, Section 2.9 concludes this chapter.

2.2 Software Risk Assessments

The SIG has developed the Software Risk Assessment method to evaluate the maintainability of a software system. A first version of this method was described more than five years ago (Deursen and Kuipers, 2003). Between 2003 and 2008, SIG experts have used this method to assess over 80 systems, almost all from industry. In the course of these assessments, the method has been refined to better suit the purpose of the SRA, which is to understand whether current costs and system risks are in line with the business and IT strategy of the client. This section contains a description of the latest version of the SRA method reflecting this experience, to show in which environment the SIG normally evaluates implemented architectures.

Goals and Deliverables The goal of an SRA is to answer the question a company has about the quality of their software system(s). Typical examples for the need of an SRA include package selection, quality assurance or deciding whether to maintain or rebuild a given system. A more detailed description of these scenarios is given by Deursen and Kuipers (2003).

The outcome of an SRA is a report containing objective measurements of the source code, an objective representation of the concerns of the business and an expert assessment of the relation between the measurements and the concerns. Lastly, a set of scenarios for reducing the impact of potential risks is given. The duration of the project typically ranges between six and eight weeks.

Roles Figure 2.1 illustrates the different roles and responsibilities in the SRA process. The SRA Consultant is responsible for the overall process and delivering the final report. The SRA Analyst assists the SRA Consultant, mainly on the technical level, and is responsible for running the source code analysis, interpreting its outcome and supporting the SRA Consultant during technical interviews. The SRA Client is the organization that requested the SRA to whom the final report is delivered. The System Client is the organization that is using, or is going to use the system. In

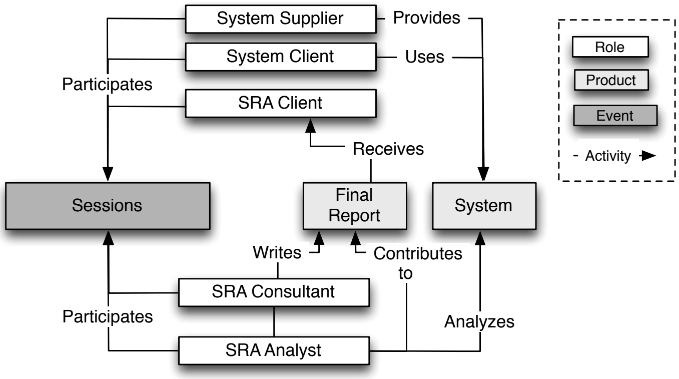


Figure 2.1: *Roles and communications within the SRA process*

most cases, the System Client and the SRA Client are the same. Lastly, the System Supplier is the organization that has developed/maintained the system.

Sessions The SRA process includes four different sessions followed by the delivery of the final report. Before starting the first session the System Supplier transfers a copy of the source code of the system, as well as available documentation, to the SRA Analyst.

The first session is the *Technical Session* and is attended by the SRA Consultant, the SRA Analyst and the System Supplier. Within this session the process and target of the SRA are explained. Additionally, this session focuses on collecting all relevant technical information of the system.

After the Technical Session, the SRA Analyst starts the extraction of source-code facts from the system. Simultaneously, the SRA Consultant conducts a *Strategy Session* together with the System Client and the SRA Client in order to precisely identify the business goals of the client.

In the *Validation Session*, the SRA Consultant, assisted by the SRA Analyst, presents the derived facts to the System Supplier and the System Client. This session provides the opportunity to identify errors in the retrieved facts.

In the last step, the SRA Consultant and the SRA Analyst map the source-code facts onto the concerns of the business and derives scenarios that confirm or mitigate the concerns. Additionally, the scenarios for reducing the impact of the risks are identified. All of this is written down in the final report which is presented in the *Final Presentation*. After this session, the final report is delivered to the SRA Client.

Experience The described SRA method has been successfully applied in the past four years. An internal report about customer satisfaction shows that customers of SIG are highly satisfied with the outcome of their SRA. The survey over the year 2008 (with a response rate of 60 percent) reveals that over 90 percent of the clients are definitely interested in a new SRA (giving it a rating of four out of five). Also, almost 80 percent of the clients would definitely recommend the SRA service to others (a number rated important by Reichheld (2003)). Additionally, we have seen a steady growth in the number of SRA's carried out in the last two years. This increased demand for risk assessments justifies a further investment in making the process more systematic, which is one of the goal of the present chapter.

2.3 Architectural System Properties

As part of an SRA, a software system is evaluated on a number of system properties, including both code-level properties and architectural properties. Code-level properties include the *volume* of the system, the *complexity* of its units, the degree of *redundancy* in its code lines etc. These code-level properties can be measured in a fairly direct way, by gathering source code metrics, aggregating them, and comparing them to statistically determined thresholds. A more detailed description of SIG's measurement model for code-level system properties can be found elsewhere (Heitlager et al., 2007).

To evaluate the implemented architecture of the system, three architectural system properties are distinguished, corresponding to different, but complementary, perspectives.

High-level Design The architectural property of high-level design is aimed at the technical division of the overall system into layers or other organizational and/or technological components. A typical example is the division of the system into a data layer, a business logic layer, and a user interface, following the *three-tier* architectural style.

Modularization The architectural property of modularization concerns the division of the main technical building blocks into functional components. A typical example would be components for account management, interest calculation, payment processing, annual reporting, etc. A single functional area is often addressed by several related components, situated in distinct technical layers.

Separation of Concerns The architectural property of separation of concerns deals with the division of tasks over the components within layers and over the source code units within components. For example, within the component for payment processing, the tasks of user authentication, input validation, transaction logging, etc. may be addressed separately or in a tangled fashion. Also, some tasks may be handled fully at the data layer, while others are handled by a combination of units at the data and business logic layers.

Thus, these three architectural properties cover organizational elements at increasingly higher degrees of granularity: layers, components, and tasks or concerns. At the granularity of high-level design, the focus is on technological choices. At the granularity of components, the functional break-down takes center stage. Finally, at the granularity of concerns, the interplay of technical and functional divisions is addressed.

Unlike code-level properties, the architectural properties are not evaluated on the basis of source code metrics alone. Though certain source code metrics may be considered by the evaluator, many other factors are taken into account that are not readily quantified. In fact, the evaluation requires interpretation of a wide variety of observations and extensive software engineering expertise. In the remainder of this chapter, we delve deeper into the exact criteria that are applied for this evaluation.

2.4 Problem Statement

The criteria employed by SRA Consultants for the evaluation of implemented architectures have emerged from practice. The overall distinction between High-level Design, Modularization, and Separation of Concerns emerged early and has been used in a stable fashion throughout many years. However, the observations to underpin judgements about these architectural properties were selected and used on a per-evaluation basis. The SRA Consultants may share a common understanding of observable system attributes and how they influence architectural properties, but this common understanding has not been documented in an evaluation-independent and re-usable form.

The lack of a documented set of observable attributes leads to a number of limitations. Firstly, without documentation, the evaluation method can only be taught by example to new SRA consultants, which is a time-consuming process. Secondly, the structure of the argument that backs up each evaluation must be constructed from scratch each time, even though they follow the same pattern. In practice, previous arguments are used as templates for new ones, while it could be more efficient to refer to a common model. Thirdly, a documented set of relevant system attributes would augment the traceability, reproducibility, and evaluator-independence of the evaluation method. Lastly, to use the architecture evaluation results for comparing systems, e.g. in order to benchmark the architecture of a system under evaluation against the architectures of previously evaluated systems, a documented and shared overview of criteria is indispensable.

In order to discover and document a set of observable system attributes that can be used for evaluating implemented architectures, we have conducted an empirical study (Wohlin et al., 2000) into the evaluations performed by SRA Consultants of the SIG over several years. In particular, we set out to find answers to the following research questions:

Q1 Which set of system attributes do experts normally take into account when determining the quality rating of the three architectural system properties?

Q2 How do these system attributes influence the architectural system properties?

The answers to **Q1** documents which observable system attributes are relevant for architectural evaluation, while the answer to **Q2** documents which properties are influenced by them. Together, the answers to these questions help to remove the above-mentioned limitations.

2.5 Empirical Study

2.5.1 Design

The input for this study are the final reports of 44 SRA's conducted between December 2006 and August 2008. Older reports do not consider the Maintainability Model and are therefore not taken into account. The reports contain a total of 54 system ratings and are written by seven different SRA Consultants. The reviewed systems cover a wide range of languages, sizes, ratings and business areas. An overview of this data is given in Figure 2.2. Note that the two lower bar-charts respectively show the number of systems with a specific rating and the number of systems of a specific size.

Based on the guidelines as proposed by Wohlin et al. (2000) we define the following collection procedure. For each report, we extract the arguments used for the quality rating for each of the architectural system properties. These arguments can be extracted from a table that appears in most final reports. This table lists all the system properties from the Maintainability Model, the rating for each system property, and a small argumentation for this rating. Additionally, each system property is discussed in a separate paragraph in the appendix of the report.

When there is no table we only use the information extracted from the discussion paragraph. In case of ambiguity we let the arguments in the table take precedence because these are the arguments most likely used to determine the final rating. From the list of all arguments we extract the set of system attributes by examining which system attributes are mentioned in the arguments. The result of this first step is given in Section 2.5.2.

After mining the list of system attributes, we iterate through all the reports a second time. In this iteration we determine which system attributes are used to rate each of the three architectural system properties. This is done in a separate iteration because the first step has given us a stable set of system attributes to work with, which makes it easier to categorize all arguments consistently. The result of this second step is given in Section 2.5.3.

Validation of the results is done in two ways. First, we conduct interviews with two experienced SRA Consultants in which we ask for an explanation of how they

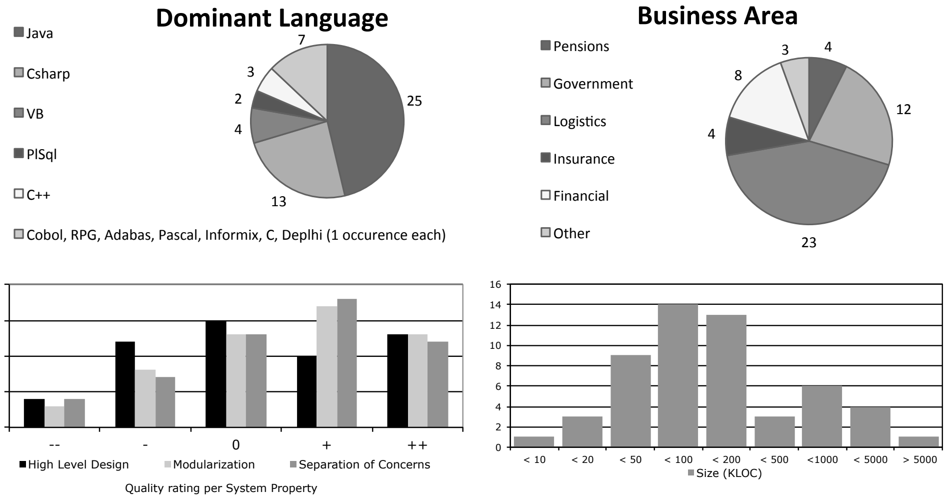


Figure 2.2: *Distribution of the key characteristics of the 54 subject (sub)-systems*

usually evaluate the three architectural system properties. Secondly, we present our findings to a group of ten SRA Consultants. In both cases the authors are not amongst the SRA Consultants. During the validation the SRA Consultants can identify new system attributes or projections. When this is not the case we conclude that the results are valid and provide a good overview of the current practice. The interview process and the reports of the interviews are described in Section 2.5.4. The results of the study are validated in Section 2.5.5.

2.5.2 Report Study Results

In order to extract the system attributes from the arguments used in the reports we used an iterative process. The first report provided us an initial set of system attributes, after which we tried to place the arguments used in the second report under these system attributes. When an argument could not be placed under an existing system attribute we introduced a new system attribute based on a general description of the used argument. Adding a new system attribute was done conservatively in order to keep the list of system attributes manageable.

Finding the system attributes used in an argument was in most cases straightforward. For example, the argument “Usage of many different technologies” clearly touches upon the *Technology Combination* attribute. On the other hand, the argument “Implementation of data-access logic is bad” does not directly mention a system attribute. After reading the accompanying paragraph it became clear that the code for data-access was scattered all over the system. Therefore, this argument touches upon the *Functional Duplication* and the *Component Functionality* attribute.

Name	Description	Assessment Approach
Abstraction	How well are input, output and functionality shielded throughout the system.	Inspecting maximum Inheritance Depth, create a call-graph showing the path between user interface and back-end.
Functional Duplication	The amount of functional duplication within the system.	Browsing the source code, identifying chunks of duplicated functionality.
Layering	The functional decomposition of the system into layers.	Inspecting the call graph on component level.
Libraries / Frameworks	The usage of standard libraries and frameworks.	Inspecting the list of imports and structure of the source- and build-files.
Logic in Database	The encoding of business logic in the database.	Inspecting the size and complexity of stored procedures and triggers.
Component Dependencies	The static dependencies (i.e., calls, includes) between components.	Inspecting the call-graph on component level, matching this against expected dependencies.
Component Functionality	The match between the expected and encoded functionality within a component.	Expected component functionality is determined by interviews and available documentation, encoded functionality is determined by browsing the code.
Component Inconsistency	Whether similar components have a different type of set-up.	Inspecting the structure of the source / method calls within a component.
Component Size	The match between expected size of a component and the actual size.	Expected component size is determined by the encoded functionality, actual size is measured by summing the LOC of all files in a component.
Relation Documentation / Implementation	The correctness of the relationship between the available documentation and the source code.	Manual inspecting of both the source code and the documentation.
Source Grouping	The complexity of grouping sources into components.	Creation of filters to put sources into components.
Technology Age	The age of the used languages and platforms.	Finding the technologies used is done by inspecting the different types of source code. Used platforms are determined by reading the documentation and through the technical session.
Technology Usage	Adherence to coding standards, patterns, and best practices.	Browsing the source code, using language specific style-checkers.
Technology Combination	How well the combination of technologies is expected to work.	Finding of the technologies is done in the same way as *Technology Age*, how common the combination is is based on expert opinion.
Textual Duplication	The amount of textual duplication within the system.	Checking the values of a duplication report.

Table 2.1: *System Attributes mentioned in the rating of properties*

	High Level Design	Modularization	Separation of Concerns
Abstraction	8	3	2
Functional Duplication	2	6	18
Layering	28	1	20
Libraries / Frameworks	22	1	1
Logic in Database	1	1	3
Component Dependencies	7	11	6
Component Functionality	4	32	13
Component Inconsistency	0	1	0
Component Size	1	1	0
Relation Doc. / Impl.	2	3	0
Source Grouping	0	14	2
Technology Age	13	0	0
Technology Usage	7	3	0
Technology Combination	5	1	0
Textual Duplication	0	0	4

Table 2.2: *Number of times a system attribute is named in the rating of a system property*

Using this process we have identified 15 system attributes that are used in the evaluation process. The list of found system attributes is given in Table 2.1 and includes items one would typically expect such as layering or the use of frameworks, as well as less common attributes such as the (un)likelihood of certain technology combinations (e.g., Java and Pascal). For each system attribute we provide a name, a definition and an operational procedure to quantify the attribute called an “assessment approach”.

2.5.3 Projection Results

After defining the set of system attributes we examined the reports in a second iteration and determined which system attributes are used as an argument for which system property. The result of this survey can be found in Table 2.2. Note that several system attributes can be mentioned in the rating of each of the system properties, which can result in more than 54 system attributes per system property.

2.5.4 Interview Description

The interviews with the two SRA Consultants took place on two different occasions. In both cases the SRA Consultant was asked to explain how he usually determines the rating of the three architectural system properties. Since the goal of these interviews is to validate our findings we did not provide the list of system attributes. Even though we did not impose a time-limit both interviews took around 60 minutes to complete. The reports of the two interviews are described below.

Expert 1 The first expert normally uses the different dimensions of the board of a tic-tac-toe game as an analogy to explain the differences between the architectural system properties to the management of an organization. He models each architectural system property as a separation of the functionality along one of the axis in the game.

Modularization is explained as the *vertical* separation of functionality. The expert looks for components in the code based on e.g. the directory structure, naming convention of files, packaging structure, etc. Roughly speaking, four components are usually expected for a system with < 20 KLOC, up to 10 components for a system < 100 KLOC and up to 20 components in larger systems. After this, the expert inspects the files in the components to discern if these components encode certain functionality in a consistent manner. The expert inspects the size of the components to see whether the distribution of the code is expected given the functionality encoded in the component, or whether there is an indication of poorly chosen components (e.g. 1 component of 10 KLOC and 15 components of 100 LOC).

Separation of Concerns is explained as the *horizontal* separation of functionality, typically encoded by the layering of the system. The expert asks questions such as: “are there layers for specific purposes such as presentation, data-access and business-logic?” and “is there one and only one place where communication with external systems or with the database is handled?”. In addition, the expert takes into account framework usage and violations between layers to determine the rating of this characteristic. Finally, the expert considers the interweaving of, for example, the definition of SQL-code and business-logic or embedding Java in JSP to have a negative impact on Separation of Concerns.

High Level Design is explained as the *diagonal* separation of concerns. The expert usually measures this by inspecting the call-graph on component level and determining the absence or presence of loops (so each dependency between components is unidirectional). In addition, the expert takes into account the usage of modern programming languages and platforms to determine the rating for this system property.

Expert 2 The interview with the second expert revealed the following definitions of the different architectural system properties. For High Level Design, the expert examines the interaction of the system under assessment with other systems. A typical question here is: ”is there a clearly defined communication channel to the outside world?” Also, the expert looks for a high level division of the system into layers with separate functionality. Furthermore, the expert inspects the relation between

	High Level Design	Modularization	Separation of Concerns
Abstraction			E2
Functional Duplication			E1
Layering	E2		E1
Libraries / Frameworks	E2		E1, E2
Logic in Database			
Component Dependencies	E1	E2	
Component Functionality	E2	E1, E2	E2
Component Inconsistency		E1	
Component Size		E1	
Relation Doc. / Impl.	E2	E2	
Source Grouping		E1	
Technology Age	E1		
Technology Usage			
Technology Combination			
Textual Duplication			

E1 = mentioned by expert 1, E2 = mentioned by expert 2.

Table 2.3: *System attributes used per system property as mentioned by the interviewees*

the provided documentation (if any) and the source code. This relation is usually given a low priority except when there are large differences. Lastly, when frameworks contribute to the layering (for example frameworks for dependency injection or persistence) the usage of these frameworks is taken into account when the expert determines the rating.

The expert rates Modularization by inspecting the way the system is divided into logical components. This division is based on the package or directory structure, interviews with the customer and the presence of clearly defined subsystems. Again, the relation between the documentation and the component structure is inspected by the expert. Given the components, the expert makes an effort to put each component into one of the layers of the system based on the functionality of the component. This also includes the division of components into functional components and components that act as utility-repositories. Finally, the dependencies between the components is assessed by inspecting the call-graph. According to the expert, a good call-graph shows the layering where each component is part of one layer and each layer depends on one lower layer. Furthermore, a good call-graph shows all the utility components because these components only receive calls. A bi-directional dependency in the call-graph usually hints at an implementation or design flaw.

For Separation of Concerns the expert inspects the separation of functionality within components. An example of this is whether the interfacing between two modules in a component is separated from the implementation. A different example is whether a component that allows access to the outside worlds implements this access as a thin layer on top of ’real’ functionality instead of encoding business logic into the functions / objects that provide the actual access. In addition, the expert takes into account frameworks that do not directly contribute to the layering of the system.

2.5.5 Validation

When we process the reports of the interviews with the experts in the same we as we analyzed the final reports, we see that they do not introduce any new system attributes. All of the arguments used for each system property can be placed under the 15 system attributes listed in Table 2.1. For example, the first expert explains that Modularization is judged by looking at the sizes of the components, which corresponds with the *Component Size* attribute. A second example is that he mentions “. . . the use of modern programming languages . . . ” as an argument for High Level Architecture. This corresponds to the *Technology Age* attribute. The experts opinion of how the system attributes project onto system properties is summarized in Table 2.3.

Additionally, the presentation of the findings to a group of ten SRA consultants did not lead to an addition of new system attributes. During the discussion that followed the presentation the SRA Consultants concluded that they did not miss system attributes they normally use. Also, they agreed that the projection of the system attributes as given in Table 2.2 provides a general overview of the current practice.

2.6 Answers to research questions

> **Q1: Which system attributes do experts take into account when evaluating architectural system properties?**

The 15 system attributes presented and defined in Table 2.1 are taken into account for the evaluation of architectural system properties by SIG’s software assessment experts.

Note that some of these attributes, e.g. *Component Inconsistency* and *Component Size*, occurred with a very low frequency (see Table 2.2). Still, these attributes were also mentioned in the expert interviews (see Table 2.3), which indicates that they are actively used and should therefore not be excluded from the list.

> **Q2: How do these system attributes influence the architectural system properties?**

The data in Table 2.2 provides the raw historical data of how the system attributes have influenced the architectural system properties. From this data, we can mathematically deduce a) which system attributes are most important for each system property, and b) which system property each system attributes influences most. The answers to these questions provides an overview of how the system properties are influenced by the system attributes.

	High Level Design	Modularization	Separation of Concerns
Abstraction	X		
Functional Duplication		X	X
Layering	X		X
Libraries / Frameworks	X		
Logic in Database			
Component Dependencies	X	X	X
Component Functionality		X	X
Component Inconsistency			
Component Size			
Relation Doc. / Impl.			
Source Grouping		X	
Technology Age	X		
Technology Usage	X		
Technology Combination			
Textual Duplication			

Table 2.4: *Most Important System Attributes Per System Property*

a) Which system attributes are most important for each system property? When all system attributes are of equal importance for each system property they would have been used a roughly equal number of times. By first calculating the average number of usages for each system property, we can filter out the system attributes with a lower usage count than this average. This filtered set gives us an overview of which system attributes are most important for that system property.

For example, when we add all mentioning of system attributes for *Modularization* we get a total of 78 usages. When all system attributes would contribute to *Modularization* in the same way we expect each system attribute to be used $78/(number\ of\ system\ attributes) = 78/15 = 5.2$ times. Since this is not the case, we can filter out the most important system attributes for *Modularization* by stripping away all system attributes which where mentioned less than five times. This leaves us with only the four most important system attributes for *Modularization*, see Table 2.4.

Comparing the projection of the specific experts shown in Table 2.3 with the projection of the historical opinion in Table 2.4 we observe that some deviation between the two exists. We assume this deviation stems from the fact that the projection of the experts is extracted from a single free-form description of the assessment process, while the projection in a report contains the consolidation of multiple discussions amongst SRA Consultants. Also, during the interviews the SRA Consultants most likely mentioned only those system attributes that they consider to be important in general, which are not necessarily the same as the system attributes that are most important for each system property.

	High Level Design	Modularization	Separation of Concerns
Abstraction	X		
Functional Duplication			X
Layering	X		X
Libraries / Frameworks	X		
Logic in Database			X
Component Dependencies		X	
Component Functionality		X	
Component Inconsistency		X	
Component Size	X	X	
Relation Doc. / Impl.	X	X	
Source Grouping		X	
Technology Age	X		
Technology Usage	X		
Technology Combination	X		
Textual Duplication			X

Table 2.5: *Most Important System Properties Per System Attribute*

b) which system property is influenced most by each system attribute? In order to answer this question, we again compute a threshold for average use, but in this case we use this threshold to filter out the most important system property for each system attribute. For example, the system attribute *Technology Combination* is used a total of 6 times. This provides us with a threshold of $6/(number\ of\ system\ properties) = 6/3 = 2$. Using this threshold we conclude that the system property *High Level Design* is most relevant for this system attribute. The algorithm described above results in the listing for which system property is influenced most by each system attribute shown in Table 2.5.

2.7 Discussion

Applicability By documenting the system attributes used to evaluate implemented architectures, as well as their projection onto system properties, the limitations mentioned in Section 2.4 are partially lifted. Training new consultants becomes easier because the documentation is available, argumentation of the evaluation can refer to this documentation which increases efficiency, and by using the documentation as a guideline during evaluations the traceability and reproducibility of the evaluation method increases. Lastly, ratings determined with the documentation in mind allow comparison against a benchmark of earlier evaluations, providing better insight into the quality of the implemented architecture under review.

In general, we can think of other useful applications for both the list of system attributes and the projection onto system properties. For example, the list of system attributes can directly be used in existing architectural evaluation methods such as the ATAM (Clements et al., 2002) to make these methods more operational.

More concretely, we envision that the list of system attributes combined with the projection can be turned into a lightweight "sanity check" for implemented architectures. This can be done by providing questionnaires with a few qualitative questions about each system attribute. When multiple persons familiar with the system fill out these questionnaires the general opinion about the quality of the implemented architecture can quickly be determined by averaging the answers. Naturally, the results of such a sanity check are not of the same quality as a complete architecture evaluation. However, performing such a check on a set of system can be useful to, for instance, filter out the system that is in most dire need of a complete evaluation. A more in-depth discussion of this topic is given in Chapter 4.

In any case, the list of system attributes serves as a "wish"-list for researchers to develop automatic, qualitative measurements. Taking into account the importance of the system attributes determined by the projection, we can quickly spot for which system attribute there is a need for qualitative, easily (or more preferably automatically) calculable metrics. Developing and validating these type of metrics is done in Chapter 6 to Chapter 9.

Threats to Validity A first threat to validity is whether the data sources of the empirical study are representative. The data in Figure 2.2 shows that the used reports were written for systems covering a wide range of industries, system sizes and programming languages. Thus, the study was not focussed on only a single type of system. Moreover, the lower-left chart in Figure 2.2 shows that the 54 ratings cover the complete spectrum of quality ratings. This implies that the study is not based on only problematic systems. Based on these two properties we conclude that the data sources of the study are representative sample of industry systems.

The data extracted from the interviews might not be representative because we have only interviewed two SRA Consultants. Even though the two interviewed SRA Consultants are amongst the most experienced consultants within SIG, the low number of interviewees might lead to system attributes not being discovered through these interviews. This threat is countered by the fact that we used the interviews only as a secondary source of validation.

A second threat is the reliability of the measurements of the report study. Even though a consistent process was followed, a different person might find different arguments in the reports, possibly leading to a different set of system attributes. This threat is countered by the presentation of the system attributes to a group of 10 SRA consultants.

Finally, a threat to validity that needs to be discussed is whether the results can be generalized towards an environment outside of SIG. In other words, can we infer that the system attributes are also important in evaluations carried out in other contexts? This threat is partly countered by the fact that the results of the evaluations based on these system attributes have been accepted by the system supplier. Based on this we argue that other evaluation contexts should find these system attributes important as well. To allow others to validate or refute this claim we have ensured that the description of our empirical study allows replication of this study in other contexts.

2.8 Related Work

In a study done at AT&T (Avritzer and Weyuker, 1998), fifty evaluation reports were mined for indicators that can predict the *risk* of a project. The authors found twelve main categories of issues. Unfortunately, these twelve categories where too broad to be useful, so they had to use more concrete issues to make a reliable method for predicting the risk of a project. Even though the domain of this study (risk of a project) is different from the domain of our study (maintainability of a product), it does show that it is useful to identify lower-level system attributes.

As discussed in Section 2.7, the list of system attributes together with the analysis of the projection provides us with a first overview of criteria for evaluating implemented architectures. According to the overviews of Babar et al. (2004) and Dobrica and Niemelä (2002) there exists only a limited set of architectural evaluation methods which are specifically aimed at implemented architectures. These methods, and the methods targetting the designed architecture as well, normally include a phase in which the criteria for evaluation are defined. Our work improves on this situation by explicitly defining such a set of quality criteria, thus making it usuable within each of these evaluation methods when maintainability is in scope.

More recently, Lilienthal (2009) defined a model to assess the complexity of implemented architecture. The system attributes she uses are similar to ours, but the model does not take into account systems implemented in multiple languages. Also, the environment of the system (e.g., libraries used, platforms it runs on) are not considered. Lastly, the research presented in this chapter goes beyond the work of Lilienthal by providing not only an overview of criteria, but positioning it in a larger process to assess the maintainability of systems in general.

There also exists some research in the area of assessing the individual system attributes. For example, Lindvall et al. (2003) links the documented architecture of a system to the actual implementation. Their case study shows an example of how the connections between components, a specific instantiation of our component dependencies, are used to assess this link. As mentioned before, both Kazman and Carrière (1999) and Murphy et al. (1995) use component dependencies to assess an implemented architecture. Even though this research provides a solid basis for assessing these attributes in isolation Table 2.1 shows that they cover only a part of the aspects of an implemented architecture.

2.9 Conclusion

This chapter describes our steps for finding criteria for the evaluation of the maintainability of implemented architectures. The main contributions of this chapter are:

- A description of an empirical study using over 40 SRA reports (Section 2.5)
- The identification of 15 system attributes that have an impact on the maintainability of an implemented architecture (Table 2.1)
- An analysis of the projection of the found system attributes onto three architectural system properties (Section 2.7)

Additionally, we have extended the work presented by Deursen and Kuipers (2003) by giving a more detailed description of the SRA process in Section 2.2. Combining the identification of the system attributes with the analysis of the projection provides us with a first overview of concrete criteria for the evaluation of the maintainability of implemented architectures.

While the system attributes presented in this chapter are based on experience taken from practice, the next chapter focusses on validating these system attributes using theories taken from the field of cognitive science.

CHAPTER 3

A Cognitive Model for Software Architecture Complexity [*]

Abstract

Evaluating the complexity of the architecture of a software system is a difficult task. Many aspects have to be considered to come to a balanced assessment. Several architecture evaluation methods have been proposed, but very few define a quality model to be used during the evaluation process. In addition, those methods that do introduce a quality model do not necessarily explain why *elements of the model influence the complexity of an architecture.*

In this chapter we propose a Software Architecture Complexity Model (SACM) which can be used to reason about the complexity of a software architecture. This model is based on theories from cognitive science and system attributes that have proven to be indicators of maintainability in practice. SACM can be used as a formal model to explain existing quality models, and as a starting point within existing architecture evaluation methods. Alternatively, it can be used in a stand-alone fashion to reason about a software architecture's complexity.

3.1 Introduction

Software architecture is loosely defined as "the organizational structure of a software system including components, connections, constraints, and rationale" (Kogut and Clements, 1994). The quality of the software architecture of a system is important, since *Architectures allow or preclude nearly all of the system's quality attributes* (Clements et al., 2002).

[*] Originally published as a short paper in the proceedings of the IEEE 18th International Conference on Program Comprehension (ICPC 2010) (Bouwers et al., 2010). Terminology has been updated to fit this thesis.

Many architecture evaluation methods have been introduced to evaluate the quality of a software architecture (for overviews see Babar et al. (2004) and Dobrica and Niemelä (2002)). However, many of these evaluation methods do not define a notion of "quality". Usually, the methodology defines a step or a phase in which evaluators are urged to define which quality aspects are important and how these aspects should be measured and evaluated. In a sense, these aspects and their evaluation guidelines define a quality model for the architecture under review.

We defined an informal quality model in Chapter 2 in the form of 15 system attributes. Because these attributes have been extracted from practice we know that quality evaluators consider these aspects to be indicators for the maintainability of an implemented architecture. However, the list of attributes does not explain *why* these attributes influence the maintainability of the architecture.

A formal model that provides more explanation has been introduced by Lilienthal (2009). This architecture complexity model is founded on theories in the field of cognitive science and on general software engineering principles. The model has been successfully applied in several case studies. However, due to the design of the model and the case studies, the scope of this model is limited to explaining the complexity of an architecture from the individual perspective of a developer. In addition, the validation of the model is only based on systems written in a single (object-oriented) language. Lastly, the model does not provide an explanation for all the system attributes found in Chapter 2.

In this chapter, we introduce the Software Architecture Complexity Model (SACM). This model extends the architecture complexity model of Lilienthal by taking into account the environment in which a developer has to understand an architecture. SACM provides an explanation for all system attributes as defined in Chapter 2 and can be used in various situations in both industry and research.

This chapter is structured as follows. First, the scope of SACM is defined in Section 3.2. More information about the background of the creation of SACM is given in Section 3.3. Section 3.4 provides an overview of the model in terms of its goal, factors and criteria. More detailed explanations of these criteria are given in Section 3.5. In Section 3.6, we discuss whether the objectives of SACM are achieved, and what the limitations of the model are. Finally, Section 3.7 concludes the chapter.

3.2 Scope and Contribution

In the IEEE Standard Glossary of Software Engineering Terminology (IEEE, 1990), the term "complexity" is defined as:

> The degree to which a system or component has a design or implementation that is difficult to understand and verify.

Based on this definition, the complexity of a software architecture can be found in the *intended*, or *designed* architecture, and in the *implemented* architecture. The intended architecture consists of design documents on paper, whereas the implemented architecture is captured within the source code.

The main contribution of this chapter is the definition of the Software Architecture Complexity Model (SACM). SACM is a formal model to reason about 1) why an implemented software architecture is difficult to understand, and 2) which elements complicate the verification of the implemented architecture against the intended architecture.

While presenting SACM in the following sections, we demonstrate that:

- SACM is defined as a formal model based on the existing theory of quality models
- the factors and criteria that are examined by SACM are grounded in the field of cognitive science and backed up by best practice and architecture evaluation experience
- SACM is independent of programming languages and application context.

In addition, we will argue that SACM can serve as a framework for further research to discover at which point developers experience certain attributes of a software architecture as being too complex.

3.3 Background

3.3.1 Related Work

Over the years, several proposals have been made to define the complexity of an architecture in the form of metrics. The following overview is by no means complete, but is meant to give a small taste of the available work in this area. A more complete overview of complexity metrics is presented by Lilienthal (2008 & 2009).

In many cases, proposed complexity metrics are based on the dependencies between components. For example, it has been suggested to count the number of dependencies (Zhao, 1998), or to use the cyclomatic complexity of the used relations (McCabe and Butler, 1989) between components.

Other metrics defined on the level of the complete architecture can be found in the work of Kazman and Burth (1998) and Ebert (1995). Kazman defines architecture complexity as a combination of the number of architecture patterns needed to cover an architecture, and which proportion of an architecture is covered by architecture patterns. In contrast, Ebert has constructed a complexity vector based on sixteen measures which have been identified in discussions that took place during a workshop.

In addition to these types of architecture level metrics, both AlSharif et al. (2004) and Henderson-Sellers (1996) introduce intra-component metrics for complexity. Descending even lower, we can find plenty of metrics for complexity on a unit level in

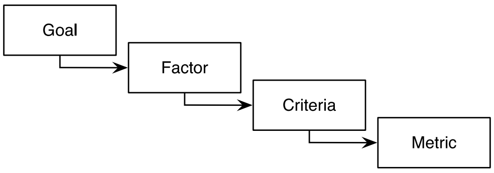


Figure 3.1: *General structure of factor-criteria-metrics-models*

for example the Chidamber and Kemerer (1994) metrics suite and the overview of Zuse (1990).

A major limitation of these metric-based contributions, but also of other discussions about software complexity such as the one by McDermid (2000), is that they provide insight into the characteristics of complexity using a single or a small set of attributes. However, none of these approaches provides a structured insight into the complexity of the software architecture as a whole. In order to provide such a complete view, a model which explains the relationship among the separate characteristics is needed.

3.3.2 Design Process of SACM

One way to provide a framework to express the relationship among metrics is to define a factor-criteria-metric-model (FCM-model) (McCall et al., 1977). An FCM-model aims to operationalize an overall goal by reducing it to several *factors*. These factors are still abstract terms and need to be substantiated by a layer of *criteria*. The lowest level in a FCM-model consists of *metrics* that are derived from the criteria. A visual representation of this breakdown can be found in Figure 3.1.

An existing FCM-model for architecture complexity can be found in the work of Lilienthal (Lilienthal, 2009). The complexity model of Lilienthal (CML) describes three factors of architecture complexity: *Pattern Conformity*, *Ordering* and *Modularization*. These factors are based upon a combination of theories from cognitive science and general software engineering principles. Each of these factors is translated into a set of criteria (for example the criteria *size of cycle* for the factor *ordering*) which are in turn evaluated using questionnaires and metrics (Lilienthal, 2008).

Unfortunately, two aspects of the design and evaluation of CML prevent it from being used in a general setting. On the one hand, the theories taken from cognitive science focus on a specific level of understanding. On the other hand, the scope of the validation of the CML is limited to systems written in only a single (object oriented) language.

The cognitive science theories underlying CML only consider understanding from within a single person. In other words, the theories only consider how understanding works inside the mind of the individual. In his book *Cognition in the wild*, Hutchins argues against the view of the person as the unit of cognition that needs to be studied (Hutchins, 1996). Instead, he argues that cognition is fundamentally a cultural

	Ordering	Modularity	Pattern conformity
Abstraction			X
Functional Duplication			
Layering	X		
Libraries / Frameworks			
Logic in Database			
Component Dependencies	X	X	
Component Functionality		X	
Component Inconsistency		X	X
Component Size		X	
Relation Documentation / Implementation			X
Source Grouping			X
Technology Age			
Technology Usage			
Technology Combination			
Textual Duplication			

Table 3.1: *Mapping of how system attributes are captured under factors of CML. An empty row indicates that the system attributes cannot be mapped.*

process. For instance, in any society there are cognitive tasks that are beyond the capabilities of a single individual. In order to solve these tasks, individuals have to work together using a form of *distributed cognition*. The interpersonal distribution of the cognitive work may be appropriate or not, but going without such a distribution is usually not an option.

Within software engineering, the cognitive task of understanding the implemented architecture of a system with, e.g., 4 million lines of code is simply too much for a single individual. Several persons need to work together in order to form a shared understanding and thus be able to reason about changes and improvements of the architecture.

Note that this widened perspective does not mean that personal factors of understanding should be discarded. Instead, a model for software architecture complexity should incorporate factors that deal with the environment in which the software architecture needs to be understood.

The case studies used to evaluate CML were all implemented in Java. Because of this, the complexity of, for example, using different programming languages inside a single system is not taken into account.

Moreover, there are more attributes that influence the complexity of a software architecture that are not considered in the original model of Lilienthal. In Chapter 2 we examined over 40 reports which contain, amongst others, evaluations of implemented software architectures. From these evaluations, a list of 15 system attributes

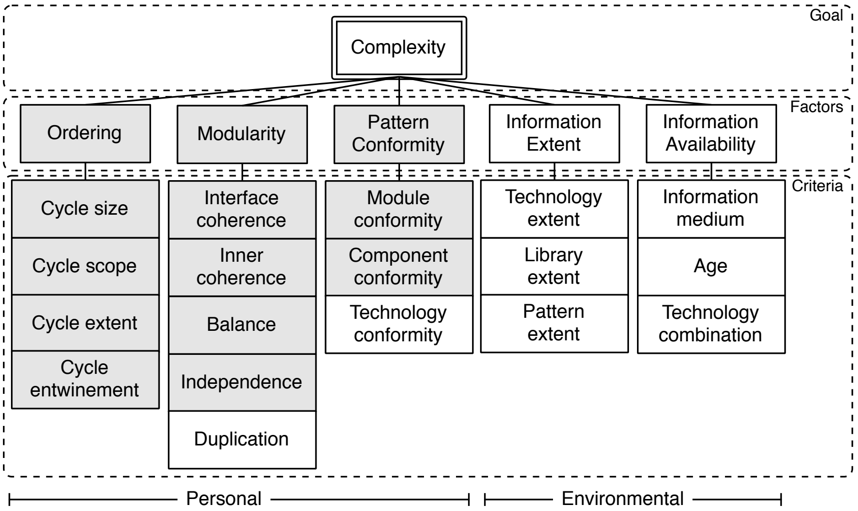


Figure 3.2: *Detailed overview of SACM, grayed factors and criteria denote the original Complexity Model of Lilienthal (2009) (CML), white factors and criteria represent our extension.*

which influence the maintainability of an implemented architecture was extracted. The list of system attributes can be found in the rows of Table 3.1.

Since a system attribute has to be understood before it can be maintained, the expectation is that all system attributes can be explained by CML. For some attributes, the mapping onto CML is straight-forward. For example, the definition of the factor "Pattern conformity" directly corresponds to the definition of the attribute "Relation between documentation and implementation". However, despite various discussions, the mapping of other attributes such as "Technology Combination" and "Technology Age" turned out to be impossible.

The final conclusion of our discussions about mapping the system attributes onto CML can be found in Table 3.1. In the end there was no explanation found for seven of the fifteen system attributes. This shortcoming has lead to the design and definition of SACM. And even though the original CML can still be found in this design, we will show that this extension is novel.

3.4 Overview of SACM

An overview of SACM is given in Figure 3.2. At the top, the overall goal of SACM is displayed. To simplify matters, this goal is named "complexity", although strictly speaking the goal of SACM is to evaluate and reduce complexity.

The overall goal is divided into five different *factors*. These factors are partitioned into two different categories. The first category are the *personal* factors. This category captures those factors that play an important role in how a single person internally understands the architecture and include small extensions on the factors from CML.

On the right, the two *environmental* factors are shown. These factors reason about the role the environment of the architecture plays in understanding an architecture. These factors are based on the theory of Hutchins (1996) and some of the system attributes that were not explained by CML.

Underneath each factor, a list of *criteria* is displayed. Those criteria with a grey background are taken from CML, while the criteria shown in white are newly defined for SACM. To keep the figure simple, the metrics and questions used to evaluate the criteria are not shown. However, examples of metrics are provided with the description of the criteria in Section 3.5. Even though the description of the personal factors and some of the criteria draw heavily from previous work (Lilienthal, 2008 & 2009), we feel that all elements should be discussed in order to completely understand the foundations and reasoning underlying SACM.

In Section 3.4.1 several basic terms are defined that are used in SACM. After this, Section 3.4.2 provides a description of the personal factors, after which the two environmental factors are discussed in Section 3.4.3. The criteria of the different factors are introduced in the next section. This separation of the description of factors and criteria makes it easier to understand the interplay between the factors, without the need to explain the criteria in detail.

3.4.1 Definition of terms

The following terms will be used throughout the description of SACM and therefore deserve a definition: *module*, *component* and *technology*.

A *module* is a logical block of source code that implements some sort of functionality. In object oriented programming languages this normally is a class, whereas in imperative languages the usual module is normally the file. To capture both paradigms, in the context of SACM a module can be defined as a source file.

A *component* is a coherent chunk of functionality within a system on a higher level than a module, such as a package in Java or a namespace in C#.

Within the context of SACM, the term *technology* includes programming languages, build tools and runtime-components such as interpreters and servers. This broad scope is chosen in order to reason about, for example, the choice a certain build tool has on the partitioning of the source code into components.

3.4.2 Personal Factors

A single developer needs to understand an existing software system and its architecture before introducing new functionality or fixing bugs. To understand the architecture, the developer needs to handle a large structure that makes up the implemented architecture, including the large number of elements on different levels (modules, components and layers) and the even larger number of relationships between those elements. In addition, the software developer has to map the intended architecture onto the implemented architecture in the source code.

To model the strategies used for this understanding, Lilienthal turned towards the field of cognitive science (Lilienthal, 2008). This area examines how human beings deal with complex contexts in their daily live. In this context, the question that is answered by cognitive science is “How do human beings understand and learn about complexity, and how do they solve problems related to complexity?”

Within cognitive psychology, three mental strategies have been described that are used by human beings to deal with large structures and complex mappings (Anderson, 2000; Norman, 1982): chunking, formation of hierarchies and the development of schemata.

Chunking refers to a strategy for making more efficient use of short-term memory by recoding detailed information into larger chunks. This strategy can only be applied successfully if the information to be chunked represents a meaningful cohesive module.

With the *formation of hierarchies*, human beings try to structure information on different levels, analogous to chapters in a book. Most of the time human beings combine formation of hierarchies and chunking to handle a huge amount of information.

Schemata on the other hand are mental structures that represent some aspects of the world in an abstract way. For example, most people have a teacher schema and can apply it to a person they have never seen before. Just mentioning that someone is a teacher instantly activates our schema and we know the job of the person, and attach certain characteristics to the person. People use schemata to organize current knowledge and provide a framework for future understanding. By organizing new perceptions into existing schemata, people can act efficiently without effort.

A number of investigations have been carried out to verify that chunking, hierarchies and schemata can help software developers to cope with large structures (Haft et al., 2005; Simon, 1996; Gamma et al., 1995; Riel, 1996; Ebert, 1995). In parallel, general design principles have been developed in the last 40 years that support these mental processes: modularization (Martin, 2002; Parnas, 1972) and abstraction through interfaces (Martin, 2002) to foster chunking, avoiding cyclic structures (Brügge and Dutoit, 2009; Henderson-Sellers, 1996; Storey et al., 1999) and layering (Cockburn, 2003; Melton and Tempero, 2007) to support hierarchies as well as design patterns and architectural styles (Gamma et al., 1995; Fowler, 2002; Evans, 2003) to facilitate schemata.

Based on the three mental strategies from cognitive psychology and the above listed general design principles, we define three personal factors for architectural complexity to be applied on all levels of abstraction (from methods to modules and to components):

- Modularity: checks whether the implemented architecture is decomposed into cohesive elements that encapsulate their behavior and offer a cohesive interface.

- Ordering: checks whether the relationships between elements in the implemented architecture form a directed, acyclic graph.

- Pattern conformity: checks whether the pattern of the intended architecture and the technology in use can be recognized in the implemented architecture, and whether their rules are adhered to.

3.4.3 Environmental Factors

In an ideal case, all parts of an implemented architecture of a system can be understood by a single person at a global level of abstraction. However, due to the size of modern software systems this cognitive task exceeds the capabilities of a single person. In addition, the speed at which the implemented architecture needs to be adapted to changing business requirements can demand several people to work together in some form of distributed cognition.

This situation is similar to the scenario of navigating a ship as described by Hutchins (1996). A single person can perform all the cognitive tasks to navigate a ship when on open sea, but navigating a ship safely into a port requires a substantial navigation team.

A reason for the need of distributed cognition is the amount of information that needs to be processed in a certain time interval. In order to lower the amount of distributed cognitive work needed, one strategy is to lower the amount of information needed to understand the implemented architecture.

Within the context of SACM, the term “information” refers to every piece of meaningful data that is needed to understand the architecture. How easy it is to understand pieces of information depends on the experiences of a specific person. This is one of the reasons why “complexity” is a subjective term. After all, an experienced Java developer will have less trouble understanding a system programmed in Java than a system implemented in Cobol. However, when more information needs to be understood, it is less likely that a single person can provide all the context for this information himself. When this happens, the context for the information has to be determined by interacting with external entities, which complicates the process of understanding and is thus more complex. Because of this, the extent of information that needs to be understood should be kept to a minimum.

In order to be able to process the needed information, it is important that the information is actually available. After all, understanding the implemented architecture of a system is virtually impossible when the implementation of the system is not available. In addition, the representation of the available data contributes to the complexity of understanding the data. For example, it is usually easier to grasp the allowed dependencies between components when these dependencies are shown in a picture, as opposed to when the dependencies are specified as a list of text.

Based on these observations, we define two environmental factors for architectural complexity: *information availability* and *information extent*, where the latter refers to the total amount of information that needs to be understood.

3.5 Criteria of SACM

Now that the factors of SACM are defined and their relation is made clear, the criteria supporting each factor can be explained. As illustrated in Figure 3.1, some of the criteria in SACM are based upon the earlier work of Lilienthal (2009). Those attributes that are not based upon CML are inspired by the criteria identified in Chapter 2.

The criteria of the personal factors are explored in Section 3.5.1, Section 3.5.2 and Section 3.5.3, after which the environmental factor are discussed in Section 3.5.4 and Section 3.5.5. We have chosen to define all criteria in full, in order to make sure that the model is self-contained.

3.5.1 Pattern Conformity

The factor *pattern conformity* checks whether the patterns that have been defined for a software can be found, and are adhered to in the implemented architecture. There are various sources of patterns in software architecture, such as design patterns, architectural patterns, best practices, coding standards and the intended architecture. Each of these patterns addresses a different aspect of the software architecture. Design patterns give guidance for the interfaces of programming modules and the interaction between them. The intended architecture and architectural patterns, such as a layered architecture, apply to the level of components in a software architecture. Finally, best practices and coding standards stem from the technology that is used.

To capture these three aspects of pattern conformity we have defined the following criteria:

1. Module conformity: checks patterns on module level

2. Component conformity: checks patterns on component level

3. Technology conformity: checks the usage of the technology against best practices and coding standards

To evaluate pattern conformity, metrics, (automated) code reviews, information from the documentation and interviews with the system's architects have to be combined. For example, the patterns defining the dependencies between components should first be extracted from the documentation or from interviews with the architects of the system. This data can then be combined with the (static) dependencies to partly evaluate the component conformity. As another example, a language specific code-style checker can be used to check the conformity on technology level.

3.5.2 Ordering

The factor *ordering* checks whether the relationships between elements in the implemented architecture form a directed, acyclic graph. This factor operates on both the module and component elements within the architecture. The first step in evaluating this factor is to check whether the graph of elements is acyclic. When this is not the case, a more detailed evaluation of the cycles must be conducted to assess the difficulty of removing the cycles from the architecture. To conduct this detailed evaluation we have defined the following criteria:

1. Cycle size: checks how many artifacts and dependencies are part of cycles in the system

2. Cycle extent: checks how many artifacts belong to the largest cycles

3. Cycle entwinement: checks how many bidirectional dependencies can be found in the cycles

4. Cycle scope: checks whether component cycles are based on module cycles

With the first criteria *cycle size* we get an indication how ordered the evaluated software architecture is. Our expert opinion state that systems with more than 10 percentage of modules in cycles tend to be hard to maintain. But if all the cycles are small, for example 2 or 3 modules, the problem is not that big. Therefore the second criteria *cycle extent* investigates the extent of the largest cycles. Still the extent of cycles gives us no final indication whether they will be hard to remove or not. The third criteria *cycle entwinement* provides us with an answer to this question. If the artifacts in a cycle are connected by several bidirectional dependencies, they are strongly linked with each other and breaking up the cycle will lead to a redesign of all the artifacts. If there is only one dependency in a cycle going against the ordering and all other dependencies are not bidirectional the case is much easier. Finally, the forth criteria *cycle scope* deals with cycles on the component level that are based on module cycles. A pure component cycle reveals the unsound sorting of some modules. If these modules are moved to other components the component cycle is resolved. If the component cycle is based on a module cycle resorting the modules will not solve the problem, but the modules in the cycle will have to be redesigned to remove the component cycle.

On the base of these four criteria a profound evaluation of ordering in software architectures becomes possible. Metrics to evaluate these criteria can be found in tools that check dependencies and in the extensive literature on graph-theory.

3.5.3 Modularity

The factor *modularity* checks whether the implemented architecture is decomposed into cohesive elements that encapsulate their behavior and offers a cohesive interface. In other words, each component in the system should have a single responsibility and should offer access to that one responsibility in a consistent way. If this is done correctly and in a balanced way, it becomes relatively easy to recognize these chunks as a whole and treat them as one block of functionality. Apart from these intra-component properties, there are some inter-component properties that are desirable to avoid confusion. Orthogonal to the fact that each component should have a single responsibility, each responsibility in the system should be encoded in a single component. This is to avoid a situation in which a change in functionality causes changing multiple components. In addition, each component should be fairly independent, in order to avoid that a change in a single component trickles through to many other components.

The criteria defined for this factor are:

1. Inner Coherence: checks whether each component has a single responsibility
2. Duplication: checks whether each responsibility is encoded in a single component
3. Interface Coherence: checks whether each component offers access to a single responsibility
4. Independence: checks how cross-linked a component is
5. Balance: checks the distribution of system size over the components

There are many metrics available to help in evaluating these criteria. Within the related work section, several metrics regarding independence have been named. In addition, metrics such as the percentage of textual and functional duplication in the system can be used to evaluate the duplication criteria. To assess the criteria of inner and interface coherence, the description of the functionality and interface of each component can be used. By combining these metrics by manual code inspection of the encoded functionality, a balanced assessment can be made.

3.5.4 Information extent

The factor *information extent* checks the amount of information needed to understand the implemented architecture. A large portion of this information need is dictated by the technologies which are used within the implementation. In order to comprehend the function each technology fulfills within the architecture, information about the semantics, the syntax and the task of each technology is needed. The more technologies are used, the bigger the total extent of information will be. For example, the information extent for a system implemented in only the Java language is smaller then a system which is implemented using Java, HTML and Javascript.

A strategy to reduce the information extent is the usage of pre-existing solutions such as libraries, frameworks and design patterns. In order to use these solutions, a developer only needs to understand *which* task the solution accomplishes, not *how* the task is accomplished. When the information needed to be able to apply a solution is lower then the information needed to encode the solution itself, the information extent is reduced.

For example, in the development of an interactive website it is a good idea to use an existing library to abstract over the implementation details of different web-browsers. The information needed to understand what the library does is less than the information needed to understand how the details are handled. On the other hand, if only a small part of a library is used, the information needed to understand what the library does might not outweigh the amount of information that it hides. Because of this, it is important to not only assess whether libraries can be used, but also to take into account the trade-off between the reduced and required information extent.

Based on these observations, we define the following criteria for this factor:

- Technology extent: checks the used technologies
- Library extent: checks the used libraries, their coverage and whether parts of the system could be replaced by available libraries.
- Pattern extent: checks the used patterns and whether certain parts of the system can be replaced by standard patterns.

In order to evaluate the first criteria, the list of used technologies, the percentage of the system which is implemented in each technology and descriptions of the tasks of each technology has to be assembled. For the second criteria, metrics such as the list of libraries used, the percentage of the usage of each library, and the percentage of functionality within the system that could be captured by an existing library have to be asserted. Lastly, metrics for the third criteria are the list of pattern definitions which are used and a description of common strategies used within the system.

3.5.5 Information availability

The aim of the factor *information availability* is to assess the availability of information about the architecture. One aspect that greatly influences the availability of information is age. Due to erosion of the information, getting up-to-date information about an old system, technology or library becomes increasingly complex. For example, locating an experienced Java developer is currently quite easy, in contrast with hiring an experienced Algol developer.

Another aspect that complicates the gathering of information is the way technologies are combined. For example, the combination of Java and Cobol in a single system is quite rare. Because of this, there is little documentation available of how this integration can be accomplished. This in contrast with finding information about using Java together with Java Server Pages.

However, apart from the fact that it should be possible to obtain information, it is also important that the information can be placed in a meaningful context. How easy it is to place information in a meaningful context is greatly dependent on the medium of the information.

For example, consider the difference between just having access to the source code of a system, or having access to a developer who has worked on the system. In the second situation, understanding the architecture becomes a less complex task because 1) the developer is capable of providing both factual information of the system and a context, and 2) it is easier to interact with the developer in order to extract information because the developer can more easily understand what you want to know.

To capture these aspects of this factor, we define the following criteria:

1. Information medium: checks the medium on which information is available
2. Age: checks the age of the used technologies, libraries and the system
3. Technology combination: checks whether the combination of used technologies is common

Examples of metrics for the first criteria are “amount of textual information”, “number of experienced developers on the development team” and “language of the documentation”. For the second criteria, metrics such as “year of first release” and “size of community” can be used. Evaluating the last criteria can be done by counting the number of times a technology is mentioned in the documentation of the other technology, or locating predefined interfaces within the technology itself.

3.6 Discussion

Section 3.2 states some desirable properties of SACM. We discuss these properties in Section 3.6.1 till Section 3.6.4. In addition, the application of SACM in industry is discussed in Section 3.6.5, after which the application of SACM in a research setting is discussed in Section 3.6.6. Lastly, a number of limitations of SACM is discussed in Section 3.6.7.

3.6.1 SACM is a formal model

Figure 3.2 shows the general design of SACM. In this figure, the overall goal of SACM is divided into several factors, which are decomposed into measurable criteria. In Section 3.5, different metrics and questions are given to support the evaluation of the individual criteria. This design corresponds to the setup of general FCM-models. Therefore, we conclude that SACM is a formal model.

	Ordering	Modularity	Pattern conformity	Information extent	Information availability
Abstraction			X		
Functional Duplication		X			
Layering	X				
Libraries / Frameworks				X	
Logic in Database				X	
Component Dependencies	X	X			
Component Functionality		X			
Component Inconsistency		X	X		
Component Size		X			
Relation Documentation / Implementation			X		
Source Grouping			X		
Technology Age					X
Technology Usage			X		
Technology Combination					X
Textual Duplication		X			

Table 3.2: *Mapping of how system attributes are captured under the factors of SACM*

3.6.2 SACM is based on both theory and practice

In Section 3.4, the relationship between the factors and existing theories from cognitive science are given. This shows that the theoretical basis of SACM can be found within this field. In addition, part of the explanation of the criteria in Section 3.5 consist of examples taken from real-world situations. Also, the grey part of the model as displayed in Figure 3.2 is mainly based on CML, which is already validated in practice. The basis of the new criteria and factors in the model are on the one hand the theories from Hutchins (1996), and on the other hand the system attributes as identified in Chapter 2. Because of this fact, we conclude that SACM is based upon both theory and practice.

3.6.3 SACM is independent of programming languages and application context

In none of the factors, criteria or metrics (excluding examples) there is a reference to a specific type of programming language or application context. In contrast, the environmental factors of SACM explicitly take into account the usage of multiple technologies within a single system. Also, the definition of the terms used in SACM given in Section 3.4.1 are designed to capture all application contexts. More generally, SACM does not make any assumptions towards the domain, the semantics or the requirements of the application. This ensures that SACM can be used as a basis to evaluate a wide range of applications.

3.6.4 SACM is novel

SACM is not a straight-forward combination of existing approaches, but rather innovates over earlier work in a number of different ways. First of all, SACM extends CML on both the factor and criteria level, see Figure 3.2. Secondly, the definition of, e.g., "Pattern Conformity" has been extended in order to be more generally applicable. Lastly, several criteria, such as "Information Medium" and "Technology Usage", are based neither upon CML, nor the system attributes. Instead, these criteria are completely new and explain the new insights obtained during our discussions about the mapping of the system attributes onto CML.

3.6.5 SACM in industry

One of our first goals was to define a formal model that could be used to explain the system attributes found by Bouwers et al. Table 3.2 shows that SACM is capable of doing this. Because of this, SACM can be used as a model inside the Software Risk Assessment (Deursen and Kuipers, 2003) process of the Software Improvement Group (SIG). This shows that there is a direct application of SACM within an industry setting.

More generally, SACM can be used to reduce the initial investment needed to start performing architecture evaluations. Since there are no external dependencies, the SACM can be "plugged into" existing software architecture evaluation methods such as the Architecture Tradeoff Analysis Method (Clements et al., 2002). Doing this does not only speed up the evaluation process, but the usage of an uniform model across evaluations allows the direct comparison of evaluation results. This is useful in, for example, the process of supplier selection.

3.6.6 SACM in research

Using SACM as a basis, we envision two different areas of further research. A first area is the development of new metrics. Currently, in order to provide a balanced assessment of some criteria, metrics need to be augmented by expert opinion. In order to lower this dependency on the opinion of experts, metrics that capture these opinions could be developed. We explore this research area for the criteria *Balance* and *Independence* in Chapter 6 to Chapter 9.

The second area of research can be found in the definition of thresholds. The basic question here is “which value of metric X indicates that the system is too complex on criterion Y?”. This question can be answered by several approaches. A first approach is statistical benchmarking. By computing the value of a metric for a large group of systems, the “normal” value of the metric can be found, everything above this value is non-standard and therefore indicates that the criterion is becoming too complex.

A different approach to determine the threshold value is by conducting experiments similar to those performed in the area of cognitive science. For example, one can try to determine the threshold value for a metric by letting subjects examine pieces of source code with different values of the metrics and ask them to answer questions related to understanding. This last approach is especially well suited to find out whether a higher value of a metric always means that the source code is more difficult to understand. Taking into account the different strategies for understanding, our hypothesis is that the statement “a higher value means more complex code” is not correct for all complexity metrics. A validation of this assumption with respect to the *Balance* criterion is discussed in Chapter 6.

3.6.7 Limitations

In order to use every part of SACM, an evaluator should have access to both the implemented and the intended architecture. If this is not possible, and only the implemented architecture is available, the assessment of, the factor “Pattern Conformity” can only be done with the criteria ”Technology Conformity”. This is because no module or component patterns are available. The remaining factors and criteria that only check the implemented architecture are nevertheless worth to be applied to achieve some results about the complexity of the implemented architecture. The same holds true for utilizing SACM to an intended architecture on paper. “Pattern conformity” can not be checked on a intended architecture alone, but many other factors and criteria will generate valuable results to judge the intended architecture.

A more fundamental limitation of SACM is that there is no formal proof to show that the model is complete. Even though parts of SACM are used in evaluating over sixty systems, it could be that some factors of complexity are not captured by the model. However, since there is also no proof that the model of “understanding” in cognitive science is complete, we accept this limitation of the model. Therefore, during the application of SACM one should always be on the lookout for new factors that can help to capture complexity.

3.6.8 Evaluation

The CML part of SACM has been validated in over 25 case studies (Lilienthal, 2008). In addition, the criteria which are based upon system attributes can build upon data of over 40 case studies (see Chapter 2). However, a study to formally validate SACM as a whole has currently not been conducted. Therefore, only anecdotical evidence of the usefulness of the complete model can be given. In order to formally support these positive first results, we plan to apply SACM in a number of formal case studies within SIG.

3.7 Conclusion

In this chapter, we introduced the Software Architecture Complexity Model. This formal model can be used to reason about the complexity of an implemented software architecture and is founded upon both theories from cognitive science and general software engineering principles. Several desirable properties of SACM have been explained, as well as how the model can be used in both industry and research.

As a next step we will use SACM as a theoretical basis for the definition a lightweight evaluation method for implemented architectures in Chapter 4. In addition, we will develop and validate new metrics to support the evaluation of the criteria of SACM in Chapter 6 to Chapter 9.

CHAPTER 4

A Lightweight Sanity Check for Implemented Architectures [*]

Abstract

Architecture evaluations offer many benefits, including the early detection of problems and a better understanding of the possibilities of a system. Although many methods are available to evaluate an architecture, studies have shown that the adoption of architecture evaluations in industry is low. A reason for this lack of adoption is that there is limited out-of-the-box process and tool support available to start performing architecture reviews.

In this chapter we introduce LiSCIA, a Lightweight Sanity Check for Implemented Architectures. It can be used out-of-the-box to perform a first architectural evaluation of a system. The check is based on years of experience in evaluating the maintainability of software systems. By periodically performing this check, the erosion of the implemented architecture as the system (and its requirements) evolves over time can be controlled.

4.1 Introduction

Software architecture is loosely defined as “the organizational structure of a software system including components, connectors, constraints, and rationale” (Kogut and Clements, 1994). Evaluating a software architecture of a system helps project teams in verifying whether the architecture complies with the design goals and wishes of the stakeholders. Additionally, the evaluation can result in a common understand-

[*] Originally published in IEEE Software, Volume 27 number 4, July 2010 (Bouwers and van Deursen, 2010). Terminology has been updated to fit the thesis.

ing of the architecture, its strengths, and its weaknesses. All of this helps a project team to determine which quality criteria the system meets, since *"Architectures allow or preclude nearly all of the system's quality attributes"* (Clements et al., 2002).

Many architecture evaluation methods are available, see for example the reviews of Babar et al. (2004) and Dobrica and Niemelä (2002). Unfortunately, as shown by a survey conducted by Babar and Gorton (2009), the adoption of architecture evaluations in industry is low. Their conclusion is that *"There is limited out-of-the-box process and tool support for organizations that want to start [architecture] reviews"* (Babar and Gorton, 2009).

In this chapter we propose a way to bridge this gap, by presenting a *Lightweight Sanity Check for Implemented Architectures* (LiSCIA). This check is based on nine years of experience in conducting structured Software Risk Assessments (see Deursen and Kuipers (2003) and Heitlager et al. (2007)), as well as our earlier research on maintainability indicators presented in Chapter 2.

LiSCIA is a concrete, easy-to-apply, architecture evaluation method of which the goal is to obtain insight in a system's quality within a day. When an evaluator applies LiSCIA at the start of a software project and then periodically, for example every six months or at every release, potential problems with the implemented architecture can be spotted quickly and dealt with it at an early stage.

4.2 Background

Existing methodologies for architecture evaluations have been divided into *early* and so-called *late* evaluations (Dobrica and Niemelä, 2002). Early evaluations focus on designed architectures, while late architecture evaluations focus on an architecture after it has been implemented. LiSCIA falls in the latter category as it is aimed at evaluating an actually implemented architecture.

Our experience shows that recurrent evaluation of an implemented architecture helps to identify *architecture erosion* (Perry and Wolf, 1992), the steady decay of the quality of an implemented architecture. In the past years, the Software Improvement Group (SIG) has been offering this type of recurrent evaluations as part of its Software Monitoring service (Kuipers and Visser, 2004) and Software Risk Assessments (Deursen and Kuipers, 2003) (SRAs). In both services, the technical quality of a system is examined and linked to business risks. In an SRA this is done once, while during the Software Monitor service the evolution of a system is followed over a longer period of time.

In Chapter 2 we describe a study which uses over 40 risk assessment reports of the past two years to identify 15 system attributes that influence the quality of an implemented architecture. These 15 attributes, together with the experience of SIG in monitoring the development of software systems in the past years, form the basis of LiSCIA. In essence, LiSCIA is designed to concretely measure these abstract attributes. Additionally, LiSCIA represents a basic formalization of the steps SIG experts normally take at the start of an SRA and during the re-evaluations within a Software Monitoring project.

LiSCIA focuses on the *maintainability* quality attribute of a software system. Due to the lightweight nature LiSCIA does not provide a complete architecture evaluation. However, by recurrently applying LiSCIA, an evaluator can obtain insights into the current status of the implemented architecture of a system. With this recurring insight, the evaluator can schedule and execute appropriate actions to control the erosion of the implemented architecture. Additionally, the result of LiSCIA offers a project team a platform to discuss current issues and can be used to justify refactorings or a broader architecture evaluation.

4.3 LiSCIA

For the design of LiSCIA, we took the following key issues into account to ensure that it is practical, yet generally applicable:

- The evaluation takes an evaluator no more than a day.
- The evaluation includes ways to improve the system, i.e., it helps the evaluator to define actions.
- The evaluation is not limited to a specific programming language or technology.
- The evaluation is able to handle different levels of abstraction.

LiSCIA is divided into two different phases: a start-up phase (done once) and an evaluation phase (performed for every evaluation). The result of the start-up phase is an overview report, which is the input for the evaluation phase. The result of the evaluation phase is an evaluation report, containing the results of the evaluation and actions to be taken. These actions might require adjustments to the overview report. Both the (possibly adjusted) overview report and the evaluation report serve as input to a re-evaluation of the system. An illustration of the complete process is given in Figure 4.1.

Before describing the two phases in depth, three key elements of LiSCIA need to be defined: the *component*, the *module* and the *container*.

4.3.1 Definitions

LiSCIA uses the module viewtype (Clements et al., 2003) to reason about the structure of an implemented architecture. This viewtype divides the system into coherent chunks of functionality called *components*. A component can represent some

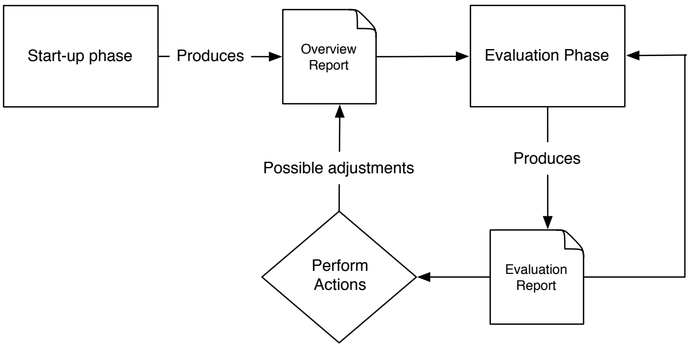


Figure 4.1: *Overall flow of LiSCIA*

business functionality, such as ”Accounting“ and “Stocks”, or a more technical functionality, such as “GUI” and “XML-processing”. LiSCIA can be applied to both decompositions.

Better yet, LiSCIA can be applied to the same version of a system using different decompositions. In this way, different modularization-views on the architecture can be explored. Each of these views can give different insights, which can lead to a better understanding of the implemented architecture as a whole.

A *module* within LiSCIA is a logical block of source-code that implements some sort of functionality. The typical module in LiSCIA is that of a source-file. Modules are grouped into *containers*; within LiSCIA the normal container is a directory on the file-system.

Using the source-file as a module complies with the notion that files are typically the dominant decomposition of functionality (Tarr et al., 1999). In other words, most programming languages use the file as a logical grouping of functionality. An argument against this decomposition is that some technologies offer a more fine-grained granularity of functionality. For example, for the Java language, the classes (or even methods) can be seen as a separate decomposition of functionality. The choice for files is made to make the method independent of the evaluated technology.

4.3.2 Start-up Phase

In the start-up phase, the evaluator first has to define the components of the system under review. In some cases, the components are defined in the (technical) documentation. In other cases, the components are apparent from the directory, package or namespace structure. When high-level documentation is not available, the components can be obtained from an interview with the developers of the system. In our experience, developers have no problem with producing a description (and drawing) of the components and the relationships between the components of the system they work on. These descriptions are often wrong in details, but adjustments to these descriptions can be defined as actions in the evaluation phase.

After defining the components, each module in the system should be placed under one of the defined components. This is done by placing patterns on the names of the modules and the containers to which they belong. For example, if all modules in the container called ”gui“ are part of the GUI-component, a logical pattern for this component would be `.*/gui/.*`.

Every module should only be matched to a single component to ensure a well-balanced evaluation. When a module should actually be placed under multiple components, it indicates that this module implements parts of different functionalities, which is a sign of limited separation of concerns. In this case, either these modules should be split up into different parts, or a component capturing the two functionalities should be introduced.

As a last step in the start-up phase, the evaluator should identify the types of technologies used within the project. For LiSCIA, the term “technologies” can be read as programming languages, but can also include frameworks, libraries, build tools and possibly even hardware platforms.

A description of the components, the patterns and a list of technologies is documented in the report of this phase.

4.3.3 Evaluation Phase

The evaluation phase of LiSCIA consists of answering a list of questions concerning the architectural elements identified in the start-up phase. Many of the questions are grouped into pairs. Usually, the first question asks for a specific situation, after which a second question asks for an explanation. The answer to the first question is either “yes” or “no”, while the answer to the second question is open-ended. This set-up requires the evaluator to be explicit, but leaves room for explaining why a certain situation occurs.

In addition to the questions, LiSCIA provides a set of actions linked to the questions. These actions can be used as a guide to answer the questions. In principle, the answers to the open-ended questions must explain why the action belonging to a question does not need to be taken. The actions are defined in such a way that when there is no valid reason to ignore the action, the maintainability of the implemented architecture can benefit from performing the given action.

The questions and actions are divided into five different categories. These categories cover the grouping of sources, the technologies used in the system, and the functionality, size, and dependencies of components. Table 4.1 lists some key-properties of these categories such as topic, number of questions and number of actions. The complete list of 28 questions and 28 actions can be found in Appendix A and online[1].

[1] `http://www.sig.eu/en/liscia`

Name	Topics of Interest	#Questions	#Actions
Source Groups	current grouping of modules in components, future components	4	4
Component Functionality	decomposition of functionality over components	5	6
Component Size	size of components, distribution of system size over components, growth of components	6	5
Component Dependencies	expected, circular, unwanted dependencies and changed dependencies	6	7
Technologies	combination, version, usage and size distribution of the used technologies	7	6

Table 4.1: *Key properties of the categories of LiSCIA*

Each category contains questions related to the current situation, as well as questions related to the previous evaluation. The latter type of questions can be ignored in the initial evaluation, since their primary objective is to reveal the reasons for differences between the versions compared.

Source Groups

A good example of the re-evaluation questions can be found in the "Source Groups" category of questions. As a first step in this category, the evaluator has to determine whether all modules belong to a component given the patterns described in the overview report. During the start-up phase, the patterns for the component are designed to capture all modules. Because of this, it is to be expected that during the first evaluation, all modules are placed under a component. Since most of the questions in this category are about modules that are not matched to a component, the questions in this category can be ignored for this first evaluation session.

During later evaluation sessions, it sometimes happens that new modules have been added to the system that are not matched by any component-pattern. We have experienced this situations on a multitude of occasions. In some cases, these modules are simply misplaced by mistake and the modules can easily be moved to their correct location. In other cases, a new component (together with a new pattern) needs to be introduced because new functionality is introduced. Questions regarding the ideas behind this new component, and possibly additional components, are also part of this category of questions.

Even though it is to be expected that the mapping of modules to components is complete during the first evaluation, this is not always the case, especially when existing documentation is used as part of the overview report. For example, in one of our assessments we evaluated a system containing over 4 million lines of code. The existing documentation described several components. In addition, the documentation included a file describing the mapping of each source-file to one of these components. Evaluating this mapping carefully, we discovered that over 30 percent of the source-files could not be assigned to a component. Additionally, a considerable part of the mapping contained source-files that did not exists anymore. One of the actions defined for this system was to re-order the source-files to better resemble the component structure as outlined in the documentation. This example illustrates that it is important to verify existing documentation against the current implementation, instead of assuming that the documentation is correct.

Component Functionality

Within the second category, the evaluator focusses on how the functionality of the system is spread out over the components. For example, one of the questions asks whether the functionality of each component can be described in a single sentence. The question helps the evaluator in determining whether the current decomposition of functionality is not too generic. In several cases, we encountered systems that defined a component called “Utilities” (or something similar). When the exact functionality of this component is described, it turns out that this component does not only contain generic functionality, but also business functionality, specific parsing functionality, or an object model. In these cases, the action is to split up the component in a truly generic part and separate components for the more specific types of functionality.

Component Size

This third category of questions is related to the size of the components. In order to keep LiSCIA usable for a large range of systems, the exact definition of “size of a component” is intentionally kept abstract. Nevertheless, in most cases the size of a component can be represented by the sum of the lines of code of all modules in the component.

The questions in this category are not only related to the size of the individual components, but also consider the distribution of the size of the complete system over the components. Additionally, this category contains questions related to the growth of the components. These last type of questions are especially useful in detecting unwanted evolution within the system, but also help in detecting unwanted development effort.

For example, in one of our monitoring projects we had several components marked as “old”. These old components contained poor quality legacy code which was still used, but which was not actively maintained. After inspection of the growth of the size of the components, it was discovered that new functionality was still being added to the “old” components. The explanation given was that it was easier to add the new functionality in this component. Even though it was easier, it also resulted in the addition of large and complex pieces of code because it had to follow the structure of the legacy-code. The action that resulted from this observation was the migration of the functionality to a new component, where it would be easier to maintain and test.

Component Dependencies

To answer the questions in this category, the dependencies between components need to be available. Similar to the size of a component, the concept of "dependencies between components" is kept abstract. However, the dependencies between components can be calculated by first determining the dependencies between modules (for example, the calls between methods inside the source-files), after which these dependencies can be lifted to the component level.

The questions in this category help the evaluator to assess the wanted, unwanted and circular dependencies between components. In addition, questions about added and removed dependencies are defined for when a previous evaluation is available. One of the questions in the latter category is whether there are any new dependencies added, as was the case on one of our monitoring projects. The follow-up question is whether this dependency is expected, which, in the case of this project, it was not. The new dependency was not allowed according to previously defined layering rules. After reporting this violation of the architecture, the project-lead was surprised and asked the developers for an explanation. The developers recognized the violation, and explained that it was introduced because a third component was not finished yet. As soon as this third component was finished, the dependency would be removed. All of this was documented in an evaluation report. Four months later, when the third component was finished, the developers were reminded of this undesired dependency and it was removed.

Technologies

The last category of LiSCIA assist the evaluator in assessing the combination of the technologies used within the system. In addition, the questions in this category deal with the age, the usage, and the support for each of the used technologies. As an example, one of the questions asks whether the latest version of each technology is used. In many projects we have seen that this is not the case. The usual explanation for this is that there is no time to upgrade the system. In these situations, an action is defined to upgrade to the latest technology as soon as possible. This is to prevent the situation in which a legacy technology is used without an easy upgrade path. A different explanation for the same situation is that management decided to always use the second-last version of a technology, because this version has already proven itself in practice. In these cases, there is no action defined, but the explanation for not implementing the action is documented.

Result

The answers to all the questions and, if applicable, a list of actions is documented in the report of this phase. With this report, certain refactorings can be justified. In addition, the report provides a basic overview of the architecture as it is currently implemented. Because of this, the report can serve as a factual basis for discussions about the current architecture. Lastly, the report is used as input to the next evaluation session.

4.4 Discussion

4.4.1 Applications

Because LiSCIA is based on our experience in monitoring existing software systems, we were able to provide examples in the last section showing how the different parts of the method can be used to discover problems in an implemented architecture. In order to evaluate the complete LiSCIA method, we are currently starting to use LiSCIA as part of our daily practice. Currently, there is not enough data available to formally evaluate the method, but the first results look promising.

LiSCIA has been applied in a number of different situations. It has been used, amongst others, to start the evaluation process of a new system and as a sanity check after an evaluation. Overall, the reactions of the consultants who used LiSCIA were positive. They appreciated the structure of the methodology, as well as the type and the ordering of the questions. One of the consultants was particularly pleased because the straight-forward application of LiSCIA has, in his words, "mercilessly led to the discovery of the embedding of a forked open source system in the code."

Apart from this discovery, the application of LiSCIA led to more detailed evaluations of certain system attributes. For example, the addition of a new technology in one system led to a detailed evaluation of how this new technology interacted with the "old" technologies, and how the old technologies interacted with each other. In a different system, the application of LiSCIA warranted a deeper look into how functionality was divided over the components. In general, LiSCIA was able to identify which attributes of the implemented architecture deserved a more thorough evaluation.

The application of LiSCIA took between about thirty minutes (for the sanity check) and one day (as the start of the evaluation of a new system). In the first case, all data needed to evaluate the system was already available, while in the latter case the data was constructed during the application of LiSCIA. In both cases, the consultants were not involved in the development of the reviewed systems.

These preliminary results show that the application of LiSCIA can be done within a day, and that LiSCIA helps to identify weaknesses in the implemented architecture. However, a more formal study needs to be conducted to confirm these results.

4.4.2 Limitations

During the initial evaluation of LiSCIA, some critical aspects of LiSCIA were brought to our attention. As a start, one of the consultants stressed that LiSCIA should be applied by an expert outside the development team, pointing out that "... it is hard for a software engineer to remain objective when it concerns his own code".

It is indeed true that LiSCIA relies heavily on the opinion of the evaluator. After all, the evaluator is the person who decides whether a given explanation is "good enough" in the setting of the project. Therefore we advise that a second expert examines the evaluation report to make sure that none of the explanations is flawed.

Different consultants also pointed out a second limitation of LiSCIA. They stated that the method relies on some of the functionality of the internal toolset of SIG, which might limit the usability outside of SIG. This is a correct observation, since LiSCIA relies on (automated) tool-support to determine the size and the dependencies between the modules of the system. However, these types of measurements can be done by freely available open source tools. The only investment needed is in the aggregation of the raw output of these tools, which is a relatively minor investment.

A third limitation of LiSCIA is the fact that it is only aimed to discover potential risks related to maintainability. Additionally, because LiSCIA uses only a single viewpoint to evaluate the architecture it is likely that it does not even cover all potential risks in this area.

In our view, these limitations can also be considered as strengths of the method. First of all, when a system is not maintainable, dealing with other quality issues such as performance and reliability becomes more difficult. Because of this, a first focus on maintainability is justified. This focus also allows for a scoped setup of the questionnaire, allowing LiSCIA to be easily implemented in current projects with little effort. Moreover, LiSCIA is deliberately positioned as a lightweight check to ensure that it is seen as a stepping-stone towards more broader architecture evaluations. Last but not least, the collection of questions and actions reflects years of experience in conducting architectural evaluations, making it likely that they cover the most important maintainability risks.

4.5 Related Work

4.5.1 Evaluating Architectures

Comparing LiSCIA against the architecture evaluation techniques mentioned by Babar et al. (2004) and Dobrica and Niemelä (2002), a major distinction can be found. In contrast to many of the available architecture evaluation, LiSCIA pre-defines a notion of quality in terms of maintainability. Other available architecture evaluation methods, including the ones explicitly aimed at an implemented architecture, do not provide such a notion. Instead, virtually all methods contain a phase in which the notion of quality should be defined by the evaluators. As discussed before, this limits the use of LiSCIA to a specific purpose, but makes it easier to start with performing an architecture evaluation.

4.5.2 Architecture Erosion

Approaches for dealing with architecture erosion typically try to incorporate a solution into the design of the architecture. This approach helps in avoiding architecture erosion, but van Gurp and Bosch (2002) conclude that *"even an optimal design strategy for the design phase does not deliver an optimal design"*. In addition, predicting all new and changed requirements during the definition of a first release of a system is impossible. So even if the design was optimal in some sense for the first release, there is a good chance the design needs to be adapted. To conclude, there is no way to completely avoid changes to an implemented architecture. Therefore, the architecture that is currently implemented should be taken into account when dealing with software change in order to avoid, or minimize, architectural erosion.

There exists approaches that do take into account the implemented architecture, for example the approach proposed by Medvidovic and Jakobac (2006). The main difference between this (and similar) approaches and LiSCIA is the time at which erosion is dealt with. LiSCIA tries to detect erosion when it has actually happened, whereas other approaches try to prevent erosion from happening. Since these approaches are complementary they are both considered to be useful and necessary. We realize that dealing with erosion after is has happened is more difficult and costly, but it is better to deal with erosion as soon as it has been introduced rather than when other issues need to be solved.

4.6 Conclusion

LiSCIA provides a lightweight sanity check to keep control over the erosion of an implemented architecture. A first introduction of LiSCIA within SIG has received positive feedback. LiSCIA is simple to apply, and therefore will not provide the same depth as a full-fledged architecture evaluation. In spite of that, the results are useful and help in detecting architecture erosion.

We are currently evaluating the formal LiSCIA method by applying it in our current practice. In addition, we are very interested to see whether LiSCIA is useful in environments outside SIG. For this, we call upon you to try out LiSCIA and share the results with us. Combining our own experience with the feedback of the community we hope to report on an improved version of LiSCIA in the coming year. The complete LiSCIA method can be found in Appendix A and online at:

`http://www.sig.eu/en/liscia`

CHAPTER 5

Getting what you measure: four common pitfalls in using software metrics [*]

Abstract

In the previous chapters the focus of our research has been on the identification of attributes which influence the maintainability of an implemented architecture, and on how these attributes can be used in a repeated evaluation of such an architecture. In the remaining chapters of this thesis the focus shifts towards the design and validation of metrics for two of these attributes, which allows the continuous evaluation of these two attributes within a software project.

Before defining new metrics we use this chapter to present four pitfalls related to the use of software metrics in a project management setting. These pitfalls are based on the professional experience of the author as a consultant and the interactions with other consultants during the course of his research. In such, they are a lightweight codification of undesired situations observed in practice and are therefore not expected to represent many new ideas. To the contrary, we expect experienced software engineers to recognize most, if not all, of these situations.

Nonetheless, by explicitly naming and describing these pitfalls evaluators working with software metrics will be able to recognize and deal with these undesired situations in a timely manner. In addition, these four pitfalls provide more context for the work presented in Chapter 6 to Chapter 9.

[*] Originally published in Communications of the ACM, Volume 55 Issue 7 (Bouwers et al., 2012).

Software metrics, a helpful tool or a waste of time? For every developer who treasures these mathematical abstractions of their software system there is a developer who thinks software metrics are only invented to keep their project managers busy. Software metrics can be a very powerful tool which can help you in achieving your goals. However, as with any tool, it is important to use them correctly, as they also have the power to demotivate project teams and steer development into the wrong direction.

In the past 11 years, the Software Improvement Group has been using software metrics as a basis for their management consultancy activities to identify risks and steer development activities. We have used software metrics in over 200 investigations in which we examined a single snapshot of a system. Additionally, we use software metrics to track the ongoing development effort of over 400 systems. While executing these projects, we have learned some pitfalls to avoid when using software metrics in a project management setting. In this chapter we discuss the four most important ones:

- Metric in a bubble
- Treating the metric
- One track metric
- Metrics galore

Knowing about these pitfalls will help you to recognize them and hopefully avoid them, which ultimately leads to being able to make your project more successful. As a software engineer, knowing these pitfalls helps in understanding why project managers want to use software metrics and help you in assisting them when they are applying metrics in an inefficient manner. For an external advisor, the pitfalls need to be taken into account when presenting advice and proposing actions. Lastly, should you be doing research in the area of software metrics, then it is good to know these pitfalls in order to place your new metric in the right context when presenting them to practitioners. But before diving into the pitfalls we first discuss why software metrics can be considered a useful tool.

5.1 Software metrics steer people

"You get what you measure"; in our experience this phrase definitely applies to software project teams. No matter what you define as a metric, as soon as the metric is being used to evaluate a team the value of the metric moves towards the desired value. Thus, to reach a particular goal one can continuously measure properties of the desired goal and plotting these measurements in a place visible to the team. Ideally, the desired goal is plotted alongside the current measurement to indicate the current distance to the desired goal.

Imagine a project in which the run-time performance of a particular use-case is of critical importance. It then helps to create a test in which the execution time of the use-case is measured on a daily basis. By plotting this daily data point against the desired value, and making sure the team sees this measurement, it becomes clear to everybody whether the desired target is being met, or whether the development actions of yesterday are leading the team away from the goal.

Even though it might seem simple, there are a number of subtle ways in which this technique can be applied incorrectly. For example, imagine a situation in which customers are unhappy because issues that are found in a product are reported, but not solved in a timely matter. In order to improve customer satisfaction, the project team is tracking the average resolution time for issues in a release, following the reasoning that a lower average resolution time results in higher customer satisfaction.

Unfortunately, reality is not as simple as this. To start, solving issues faster might lead to unwanted side-effects, for example because a quick fix now results in longer fix-times later on due to incurred technical debt. Secondly, solving an issue within days does not help the customer if these fixes are released only once a year. Lastly, customers are probably more satisfied when issues do not end up in the product at all instead of them being fixed rapidly.

Thus by choosing a metric it becomes possible to steer towards a goal, but it can also make you never reach the desired goal at all. In the remainder of this chapter we will go over some of the pitfalls that you want to avoid when using metrics to reach a particular goal. Such a goal can either be a high-level business goal ("the costs of maintaining this system should not exceed 100K per year") or more technically oriented goals ("all pages should load within 10 seconds").

5.2 What does the metric mean?

Software metrics can be measured on different views of a software system. In this chapter we focus on metrics calculated on a particular version of the code-base of a system, but the pitfalls also apply to metrics calculated on other views.

Assuming that the code-base only contains the code of the current project, software product metrics establish a ground-truth on which can be reasoned. However, only calculating the metrics is not enough. Two actions are needed in order to interpret the value of the metric: *context* should be added and the relationship with the *goal* should be established.

To illustrate these points we use the Lines of Code metric to provide details about the current size of a project. Even though there are multiple definitions of what constitutes a 'Line of Code' (LOC), such a metric can be used to reason about whether the examined code-base is complete, or contains extraneous code, such as copied-in libraries. However, to do this the metric should be placed in a context, bringing us to our first pitfall.

5.2.1 Metric in a bubble

Using a metric without proper interpretation. Recognized by not being able to explain what a given value of a metric means. Can be solved by placing the metric inside a context with respect to a goal.

The usefulness of a single datapoint of a metric is limited. Knowing that a system is 100,000 LOC is meaningless by itself, since the number alone does not explain if the system is large or small. In order to be useful the value of the metric should, for example, be compared against data-points taken from the history of the project or taken from a benchmark of other projects. In the first scenario, trends can be discovered which should be explained by outside external events. See for example Figure 5.1, which shows the LOC of a software system from January 2010 up until July 2011.

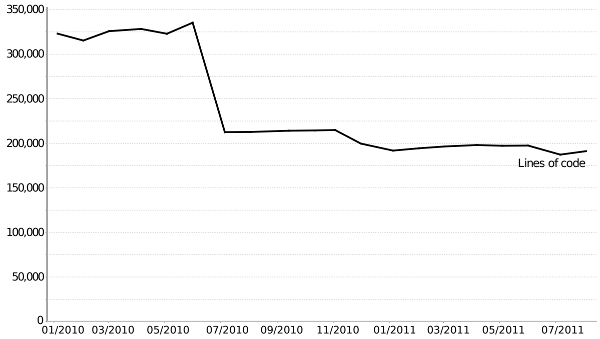


Figure 5.1: *Trendline of the Lines of Code metric over a period of 18 months*

The first question that comes to mind here is: “Why did the size of the system drop so much on July 2010?” If the answer to this question is “we removed a lot of open-source code we copied-in earlier” there is no problem (other then the inclusion of this code in the first place). Should the answer be “we accidentally deleted part of our code-base”, then it might be wise to introduce a different way of source-code version management. In this case the answer is that an action was scheduled to drastically reduce the amount of configuration needed; given the amount of code which was removed this action was apparently successful.

Note that one of the benefits of placing metrics inside a context is that it enables you to focus on the important part of the graph. Questions regarding what happened at a certain point in time or why the value significantly deviates from other systems become more important than the specific details about how the metric is measured. We often encounter situations in which people, either on purpose or by accident, try to steer a discussion towards “how is this metric measured” instead towards “what do these data-points tell me”'. In most cases, however, the exact construction of a metric is not important for the conclusion drawn from the data.

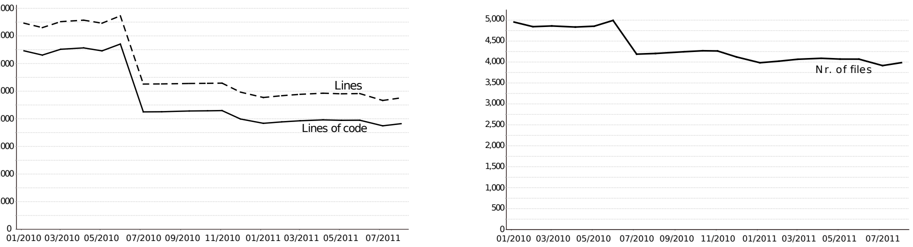


Figure 5.2: *Trendline of various volume metrics over a period of 18 months*

For example, Figure 5.2 shows plots representing different ways of computing the *volume* of a system: (1) lines of code counted as every line containing at least one character which is not a comment or white-space (Lines of Code); (2) lines of code counted as all new line characters (Lines); and (3) number of files used (Nr. of files).

The trend-lines show that, even though the scale differs, these volume-metrics all show the same events. This means that each of these metrics are good candidates to compare the volume of a system against other systems. As long as the volume of the other systems are measured in the same manner the conclusions drawn from the data will be highly similar.

Looking at the different trend-lines a second question which could come to mind is: 'Why does the volume decrease after a period in which the volume increased?' The answer to this question can be found in the normal way in which alterations are made to this particular system. When the volume of the system increases an action is done to determine whether there are new abstractions possible, which is usually the case. This type of refactoring can significantly decrease the size of the code base, which results in lower maintenance effort and easier ways to add new functionality to the system. Thus the goal here is to reduce maintenance effort by (amongst others) keeping the size of the code-base relatively small.

In the ideal situation there is a direct relationship between a desired goal (i.e., reduced maintenance effort) and a metric (i.e., a small code-base). In some cases this relationship is based on informal reasoning (e.g., when the code-base of a system is small it is easier to analyze what the system does), in other cases scientific research has been done to show that the relationship exists. What is important here is that you determine both the nature of the relationship between the metric and the goal (direct/indirect) and the strength of this relationship (informal reasoning/empirically validated) when determining the meaning of a metric.

Summarizing, a metric in isolation will not help you reach your goal. On the other hand, assigning too much meaning to a metric leads to a different pitfall.

5.2.2 Treating the metric

Making alterations just to improve the value of a metric. Recognized when changes made to the software are purely cosmetic. Can be solved by determining the root cause of the value of a metric.

The most common pitfall we encounter is the situation in which changes are made to a system just to improve the value of a metric, instead of trying to reach a particular goal. At this point, the value of the metric has become a goal in itself, instead of a means to reach a larger goal. This situation leads to refactorings to simply 'please the metric', which is a waste of precious resources. You know this has happened when one developer explains to another developer that a refactoring needs to be done because 'the duplication percentage is too high', instead of explaining that multiple copies of a piece of code can cause problems for maintaining the code later on. It is never a problem that the value of a metric is too high or too low: The fact that this value is not in-line with your goal should be the reason to perform a refactoring.

To illustrate, we have encountered different projects in which the number of parameters for methods was relatively high (as compared to a benchmark). When a method has a relatively large number of parameters (e.g., more than seven), this can indicate that the method is implementing different functionalities. Splitting the method up into smaller methods would help in making it easier to understand each separate functionality.

A second problem which could be surfacing through this metric is a lack of grouping of related data-objects. For example, consider a method which takes, amongst others, a Date-object called 'startDate' and a Date-object called 'endDate' as parameters. The names suggest that these two parameters together form a DatePeriod-object in which the startDate will need to be before the endDate. When multiple methods take these two parameters as an input it could be beneficial to introduce such a DatePeriod-object to make this explicit in the model, reducing both future maintenance effort as well as the number of parameters being passed to methods.

However, we sometimes see that parameters are for example moved to the fields of the surrounding class or replaced by a map in which a (String,Object)-pair represents the different parameters. Although both strategies reduce the number of parameters inside methods, it is clear that if the goal is to improve readability and reduce future maintenance effort these solutions are not helping you to reach your goal. It could be that this type of refactoring is done because the developers simply do not understand the goal, and thus are treating the symptoms, but we also encounter situations in which these non-goal-oriented refactorings are done to game the system on purpose. In both situations it is important to talk to the developers and make them aware of the underlying goals.

To summarize, a metric should never be used 'as-is', but placed inside a context which enables a meaningful comparison. Additionally, the relationship between the metric and desired property of your goal should be clear, this enables you to use the metric to schedule specific actions in order to reach your goal. However, make sure that scheduled actions are targeted towards reaching the underlying goal instead of only improving the value of the metric.

5.3 How many metrics do you need?

Each metric which is measured provides a specific view-point on your system. Therefore, combining multiple metrics allows you to get a well-balanced overview of the current state of your system. There are two pitfalls related to the number of metrics to be used. We start with using only a single metric.

5.3.1 One track metric

Focussing on only a single metric. Recognized by seeing only one (of just a few) metrics on display. Can be solved by adding metrics relevant to the goal.

Using only a single software metric to measure whether you are on track towards your goal reduces your goal to only a single dimension, i.e., the metric which is currently being measured. However, a goal is never one-dimensional. For software projects there are constant trade-offs between delivering desired functionality and non-functional requirements such as security, performance, scalability and maintainability. Therefore, multiple metrics need to be used to ensure that your goal, including specified trade-offs, is reached. For example, a small code-base might be easier to analyze, but if this code-base is made of highly-complex code it is still be hard to make changes.

Apart from providing a more balanced view on your goal, using multiple metrics also assists you in finding the root-cause of a problem. A single metric usually only shows a single symptom, while a combination of metrics can help to diagnose the actual disease within a project.

For example, in one project we found that the 'equals' and 'hashCode'-methods (the methods used to implement equality for objects in Java) were amongst the longest and most complex methods within the system. Additionally, there was a relatively large percentage of duplication amongst these methods. Since these methods use all the fields of a class the metrics indicate that multiple classes have a relatively large number of fields which are also duplicated. Based on this observation we reasoned that the duplicated fields form an object that was missing from the model. In this case we advised to look into the model of the system to determine whether extending the model with a new object would be beneficial.

In this example, examining the metrics in isolation would not have lead to this conclusion, but by combining several unit-level metrics we were able to detect a design flaw.

5.3.2 Metrics galore

Focussing on too many metrics. Recognized when the team ignores all metrics. Can be solved by reducing the number of metrics.

Where using a single metric oversimplifies the goal, using too many metrics makes it hard (or even impossible) to reach your goal. Apart from making it hard to find the right balance amongst a large set of metrics, it is not motivating for a team to see that every change they make results in the decline of at least one metric. Additionally, when the value of a metric is far off the desired goal a team can start to think 'we will never get there anyway' and simply ignore the metric all together.

For example, we have seen multiple projects in which a static analysis tool was deployed without critically examining the default configuration. When the tool in question contains a check that flags the usage of a tab-character instead of the use of spaces, the first run of the tool can report an enormous number of violations for each check (numbers running into the hundred of thousands). Without proper interpretation of this number it is easy to conclude that reaching zero violations cannot be done within any reasonable amount of time (even though some problems can easily be solved by a simple formatting action). Such an incorrect assessment sometimes leads to results in the tool being considered useless by the team, which then decides to ignore the tool.

Fortunately, in other cases the team adapts the configuration to suit their specific situation by limiting the number of checks (e.g., by removing checks that measure highly related properties, can be solved automatically or are not related to the current goals) and instantiating proper default-values. By using such a specific configuration, the tool reports a lower number of violations which can be fixed in a reasonable amount of time.

In order to still ensure that all violations are fixed eventually, the configuration can be extended to include other types of checks or more strict versions of checks. This will increase the total number of violations found, but when done correctly the number of reported violations does not demotivate the developers too much. This process can be repeated to slowly extend the set of checks towards all desired checks, without overwhelming the developers with a large number of violations at once.

5.4 Conclusion

Software metrics are a useful tool which offer benefits for project managers and developers alike. In order to use the full potential of metrics, keep the following recommendations in mind:

- Attach meaning to each metric by placing it in context and by defining the relationship between the metric and your goal, while at the same time avoiding to make the metric a goal in itself;

- Use multiple metrics to track different dimensions of your goal, but avoid demotivating a team by using too many metrics.

If you are already using metrics in your daily work, try to see whether it is possible for you to link the metrics to specific goals. If you are not using any metrics at this time but like to see its effects we suggest you start small. Define a small goal (methods should be simple to understand for new personnel), define a small set of metrics (e.g., length and complexity of methods), define a target measurement (at least 90 percent of the code should be simple) and install a tool which can measure the metric. Communicate both the goal and the trend of the metric to your co-workers and see the influence of metrics at work.

CHAPTER 6

Quantifying the Analyzability of Software Architectures *

Abstract

The decomposition of a software system into components is a major decision in any software architecture, having a strong influence on many of its quality aspects. A system's analyzability, in particular, is influenced by its decomposition into components. But into how many components should a system be decomposed to achieve optimal analyzability? And how should the elements of the system be distributed over those components?

In this chapter, we set out to find answers to these questions with the support of a large repository of industrial and open source software systems. Based on our findings, we designed a metric which we call Component Balance. *In a case study we show that the metric provides pertinent results in various evaluation scenarios. In addition, we report on an empirical study that demonstrates that the metric is strongly correlated with ratings for analyzability as given by experts.*

6.1 Introduction

Software architecture is loosely defined as "the organizational structure of a software system including components, connections, constraints, and rationale" (Kogut and Clements, 1994). Choosing the right architecture for a system is important, since *"Architectures allow or preclude nearly all of the system's quality attributes"* (Clements et al., 2002). Fortunately, there is a wide range of software architecture evalu-

*Originally published in the proceedings of the 9th Working IEEE/IFIP Conference on Software Architecture (WICSA 2011) (Bouwers et al., 2011a).

ation methods available to assist in evaluating an designed architecture (for overviews see Babar et al. (2004) and Dobrica and Niemelä (2002)).

After the initial design, it is important to regularly evaluate whether the architecture of the software system is still in line with the requirements of the stakeholders (Svahnberg, 2003). However, a complete re-evaluation of a software architecture involves the interaction of various stakeholders and experts, which makes this a time-consuming and expensive process. An alternative is to use more lean methodologies that support a high-frequency evaluation, such as those proposed by us in Chapter 4 and by others (Christensen et al., 2010; Harrison and Avgeriou, 2010).

Any evaluation process, the high-frequency ones in particular, greatly benefits from the use of software metrics to support it. Advantages include reducing the effort and time needed to perform the evaluation, as well as making the evaluation more objective and repeatable. Furthermore, metrics can enable continuous monitoring and thus early detection of deviations in quality.

The work on metrics for software architectures has traditionally been focussed on the way components depend on each-other and how components are internally structured (coupling and cohesion (Stevens et al., 1974; Yourdon and Constantine, 1979)). Two related aspects of a software architecture have, however, received relatively little attention when it comes to metrics: the decomposition of the system in terms of the *number of components* and their *relative sizes*.

Both of these aspects have a strong influence on how easy it is to locate the parts of the system that need to be changed, i.e., the system's *analyzability*. Having only one component (or one large component combined with several small ones) does not offer much discriminative power to locate specific functionality. In contrast, having a large number of (equally sized) components can overwhelm a software engineer with too many choices.

In earlier work, efforts have been made to quantify the relative sizes of components (Sarkar et al., 2007) and there have been references to an "ideal" number of components for a system (Blundell et al., 1997). However, there has been no effort to quantify these concerns and capture them in a single metric, such that they can be evaluated in a condensed manner.

In this chapter, we present a metric called *Component Balance* which takes into account both the number of components and their relative sizes. In order to define this metric, we determined what a "reasonable" number of components can be by studying the decomposition of over 80 systems.

To investigate whether the proposed metric accurately reflects the analyzability of a system, we performed a quantitative experiment in which we test the correlation of the values of the metric with the judgement of experts. In addition, we performed a qualitative case study on an open-source project to show that the metric is usable in an evaluation setting.

In short, this chapter makes the following contributions:

- We describe an empirical exploration of how systems are decomposed into top-level components;
- We define a metric to measure the balance of components which is usable across all life-cycle phases of a project;
- We show how the metric is correlated with the opinion of experts about the analyzability of a software system.

6.2 Problem statement

We are looking for a metric which characterizes a system's analyzability by evaluating the decomposition of a system into components. In order to define more clearly the problem at hand, we need to define our notion of *component*, as well as its connection to *analyzability*.

6.2.1 Definition of component

We define component by adopting the definition of software modules by Clements et al. (2003), i.e., a component is considered to be *an implementation unit of software that provides a coherent unit of functionality*. Within a system, such coherent units of functionality exist on multiple levels (e.g., class, package). To ensure a consistent use of the term, we define a component to be a module at *the first level of decomposition in a system*.

For example, the components could be the top-level packages in a Java system, or the collection of files which together form a working project in the IDE of a developer. A component can further be divided into modules (e.g., classes in Java or files in a working project), which are in turn decomposed into units (e.g., methods in Java or functions in C). Note that, in this definition, there is no assumption of the type of functionality which is implemented in a component. A decomposition can be based on a technical point of view (e.g., having components for file-access, network connections and the GUI) or a business point of view (e.g., containing components for savings, accounts and stocks).

6.2.2 Analyzability and component decomposition

The International Organization for Standardization (2011) standard for software quality defines analyzability, a sub-characteristic of maintainability, as: *"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"*.

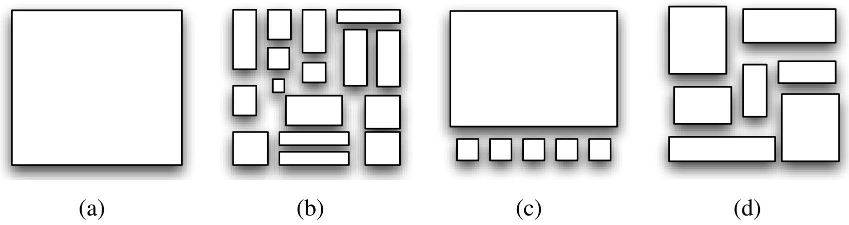


Figure 6.1: *Decomposition of three systems which is considered are hard to analyze (a), (b), (c) and one which is considered to be easy (d)*

Related to the last part of the definition, a common strategy to find those parts that need to be modified is by following the control-flow of a program. However, before this strategy can be applied, a software engineer needs to identify where to start following the control-flow for a specific feature. From a cognitive perspective, this first step is influenced by how easy it is for a software engineer to split up the overall software system into meaningful chunks of functionality (see Chapter 6, without being overwhelmed by too many choices.

To illustrate this problem, consider Figure 6.1 which shows several examples of how a system can be decomposed. Figure 6.1(a) shows the simplest case in which a system is “decomposed” into a single component. Such a decomposition hinders analyzability, as the structure of the code does not provide any hints as to where functionality is implemented. On the other hand, a division as shown in Figure 6.1(b), in which the system is decomposed into many small components, does not provide a software engineer with sufficient clues as to which component should be chosen to inspect.

However, inspecting only the number of components does not suffice to conclude about the analyzability of a system. As illustrated in Figure 6.1(c), there can still be a situation in which a system has a reasonable number of components, but where one component contains almost all the code of the system. Similar to having only a single component, this decomposition provides only limited clues as to where which functionality is implemented. To provide maximal discriminative power to a software engineer, a system should be decomposed into a limited number of components of roughly the same size, as illustrated in Figure 6.1(d).

These observations lead us to conclude that a metric which measures the analyzability of a software system must quantify whether a software system is decomposed into a *reasonable number* of components with *a low variation in size*.

6.3 Requirements

To guide our search and evaluation of a metric that fits our problem, let us establish some basic requirements. The first follows naturally from the discussion in the previous section:

R1: The metric should provide an indication of the analyzability of a software system in terms of its structural decomposition.

The fulfillment of this requirement ensures that the metric is usable during the evaluation of a software system in a single moment of time.

Apart from such a one-off assessment it is desirable to use the metric to track the evolution of the analyzability over time. To be able to compare the results of the metric over a longer time-period, we should ensure that the values of the metric are equally meaningful in every stage of the development of a software system, which leads to the second requirement:

R2: The metric should provide relevant results during all stages in the life-cycle of a software system.

Lastly, in order to ensure that the metric can be used in a wide range of systems we require that the metric is not restricted to a specific programming language or programming paradigm. This leads us to the last requirement:

R3: The metric should be technology-independent.

With these requirements in mind, let us explore existing metrics in the literature.

6.4 Related Work

One of the seven design principles of Sarkar et al. (2007) is the uniformity of component size, i.e., component should be roughly equal in size, an attribute also mentioned by several other researchers (Hatton, 1997; Hutchens and Basili, 1985; Bouwers et al., 2010). Apart from mentioning the design principle, Sarkar et al. define a metric to measure the uniformity of the component size called the Module Size Uniformity Index (MSUI) (Sarkar et al., 2007). The MSUI is defined as the division of the average component size of a system by the sum of this average component size and the standard deviation. In a later paper, Sarkar et al. specialized this metric towards object-oriented systems (Sarkar et al., 2008).

Unfortunately, the proposed metrics do not deal with all situations outlined in Figure 6.1. In particular, the situation in Figure 6.1(a) would receive the highest possible score of 1, while even the original authors of this metric state that having only a single component in a system is considered to be a bad decomposition (Sarkar et al., 2007).

In addition, both the original MSUI and its object-oriented variant are considered to be supporting metrics and are only briefly mentioned in their evaluations. In the first paper (Sarkar et al., 2007) the authors explain that the change in values is related to the way in which their clustering algorithm works, while in their second paper (Sarkar et al., 2008) the authors explain that the evaluation is not geared towards these metrics.

A second metric which can be used to measure the uniformity of the sizes of components is the Gini coefficient (Gini, 1921). This metric was proposed by the statistician Corrado Gini in 1921 to measure the inequality of the distribution of income of a given population. This measurement has recently been applied in the field of software metrics by Vasa et al. (2009) with interesting results. Unfortunately, this metric has the same problem as MSUI when it comes to dealing with a single component, and there has been no validation of this metrics with respect to measuring the size distribution of components in software systems.

Unfortunately, papers proposing metrics to address the number of components of a system are scarce. There is evidence in the literature that a single component is considered to be a bad decomposition (Sarkar et al., 2007), and that breaking up a system into many small pieces may harm reliability (Hatton, 1997). In addition, Blundell et al. conclude that there is an optimal number of components for a given software system (Blundell et al., 1997). However, we are unaware of papers which define metrics to quantify the number of components of a system with respect to this optimum or either of the extremes.

6.5 Counting components

Not having found in the literature a suitable metric for our purposes, we set out to investigate what a "reasonable" number of components might be. For that, we took an empirical approach and created a repository of different software systems. This allowed us to investigate how systems are typically decomposed into components, in particular into how many.

This section describes the general criteria for creating such a repository, the composition of our particular instance and discusses some of our observations.

6.5.1 Repository composition criteria

Establishing a repository of systems amounts to sampling individuals from a population. To ensure generalizability and reliability of the results, general considerations for sample taking should be taken to heart, such as maximizing size and representativeness, consistent data collection, and outlier inspection and removal.

Consistent data collection requires that a clear definition of *component* exists and is applied consistently across systems. Furthermore, the measurement of the volume of the various components should be done according to common guidelines or with a single tool.

The level of quality of the architecture of systems should not play a role in their selection. Otherwise a bias will be introduced in the sample which could compromise the generalizability of the results. On the other hand, the degree of stability of the architecture of candidate systems should be taken into account. Systems that are in the initial stages of development or in a phase of rapid architectural churn are best excluded from the sample, since the architecture at the time of measurement could be not representative of the architecture of the system during a substantial phase of its life.

6.5.2 Repository instantiation

Because we are interested in today's state-of-the-art, the population of software systems from which we wish to take a sample are modern, object-oriented systems that support corporations or large user communities. This excludes, for instance, old legacy systems or research prototypes. Within this group, we want systems to be represented of different sizes and development contexts (industrial and open-source).

We created a repository by gathering software systems previously analyzed by the Software Improvement Group (SIG), an independent advisory firm that employs a standardized process for evaluating software systems of their clients (Baggen et al., 2010). These industry systems were supplemented by open source systems previously analyzed by SIG's research department. Since, in the experience of SIG, the overwhelming majority of modern industrial systems are developed in C-like programming languages, we restricted our selection to Java, .NET (C#, VB.NET), and C/C++ systems.

The following table characterizes the repository in terms of number of systems per technology and development context:99

	Java	**.NET**	**C/C++**	**Total**
Industry	35	19	5	59
Open source	17	4	6	27
Total	52	23	11	86

Thus, the repository contains a total of 86 systems. Almost 70% were developed in an industrial context. About 60% were developed on the Java platform, 27% on .NET and the remaining 13% are C/C++ systems. The selected systems offer functionality in a broad spectrum of domains (e.g. public administration, finance, developer tools, system control) with a size ranging between 1 thousand and 3 million Lines of Code.

6.5.3 Component breakdown

For the industrial systems, the component breakdown was determined by the technical analysts of SIG. They work according to standard guidelines that start with elicitation of component information from the system's development/maintenance team and ends by validating the defined breakdown with that team.

For the open-source systems, the breakdown was determined also by a technical analyst and/or researcher of SIG, based on available documentation. In cases where the documentation was insufficient, the directory structure of the source-code was used to guide the component breakdown. In line with our definition of component in Section 6.2, we used only the *first major decomposition*, leaving any deeper hierarchical decomposition out of consideration. We took as leading the breakdown for the main programming language in each system, also when a deviating breakdown was present for an auxiliary language.

6.5.4 Observations

Figure 6.2 shows the distribution of the number of components per system based on the created repository. As it can be observed, the number of components metric is distributed in a non-symmetric way. In fact, a Shapiro-Wilk test (Shapiro and Wilk, 1965) yielded a p-value of 0.0014, thus allowing us to reject the hypothesis that the data is normally distributed.

A second observation is that a large portion (57%) of the systems in our repository tends to have between 5 and 10 components. If we inspect the central tendency of the repository, using the *median* in order to be robust against the asymmetry, we observe that this is valued at 8. Thus apparently, the most common number of components for a system is close to this number.

A question which this repository could answer is whether a large system more often consists of a high number of components, simply because there is more functionality to be implemented. If this is the case, a high correlation between the size of the system and the number of components in the system should be observed. When measuring the size of the systems in the repository by their Lines of Code, we observed no strong positive or negative correlation between the sizes and the number of components of a system (Spearman rank correlation shows 0.27 with a significant p-value).

We hypothesize that whenever a system grows in terms of the number of components, there comes a certain point at which the current components of the system are grouped together within a new level of abstraction. In other words, when the system grows, new levels of abstraction are added to ensure that the first level of decomposition in the system remains manageable.

6.6 Metric definition

With more empirical data on the number of components available, we can use this knowledge to define a general metric called *Component Balance (CB)*. We define this metric as a combination of two other metrics *System Breakdown (SB)*, which is designed to measure whether a system is decomposed into a reasonable number of components and *Component Size Uniformity (CSU)*, which aims to capture whether the components are all reasonably sized. We define the metrics in general, non software specific terms first, after which they are instantiated for the domain of components in a software system in Section 6.6.6.

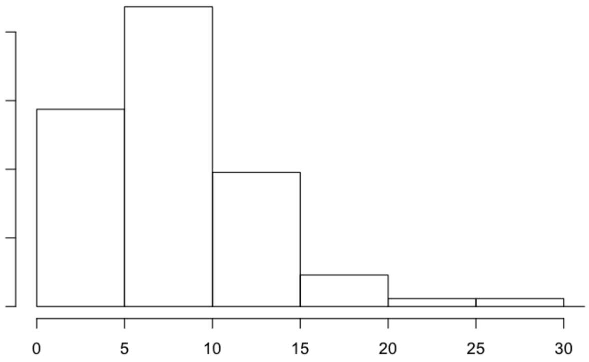


Figure 6.2: *Distribution of number of components per system*

6.6.1 Terminology

Let $S = \langle M, C \rangle$ be a system, consisting of a set of modules M and a set of components C. Each module is assigned to a component and none of the components overlap. More formally, the set $C \subseteq \mathcal{P}(M)$ is a partition of M, i.e.,

- $\forall c_1, c_2 \in C : c_1 \neq c_2 \Rightarrow c_1 \cap c_2 = \emptyset$.
- $\bigcup_{c \in C} = M$

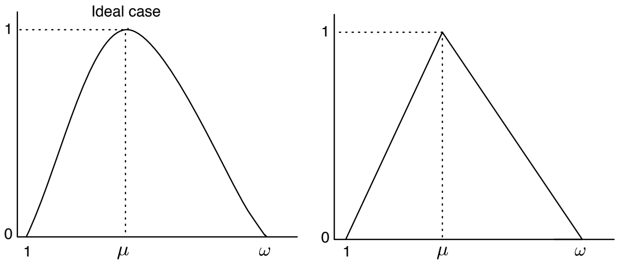
Furthermore, each module has a given size (for example measured by the Lines of Code), which is captured by a function $size : M \rightarrow \mathbb{N}$. The volume of a component $c \in C$ is defined simply as the sum of the size of its modules, thus:

$$volume(c) = \sum_{m \in c} size(m)$$

6.6.2 System Breakdown

The basis for measuring the System Breakdown (*SB*) metric is the number of components $|C|$, which is an unbounded positive number. However, a higher number of components does not imply better analyzability. As discussed in Section 6.2, both a high number of components and a low number of components hinders analyzability. To capture this in a metric, we want to map the number of components to a fixed range of numbers in which the highest number denotes a better analyzability, thus $SB : \mathbb{N}^+ \rightarrow [0, 1]$.

Since the number of components has a lower bound of 1, we define $SB(1) = 0$, thereby assigning the lowest value of the metric to the minimum number of components. On the other end, there is no theoretical upper bound for the number of components so we define an artificial upper limit $\omega > 1$ for which $SB(n) = 0$ when $n \geq \omega$. Between 1 and ω there is a number of components which depicts the best



(a) (b)

Figure 6.3: *(a) ideal component deviation function, (b) example plot of the SB function*

analyzability, let us consider $\mu < \omega$ to represent this number and define $SB(\mu) = 1$, thus assigning the highest value of the metric to the optimal number of components.

The values of the function between 1 and μ and between μ and ω are still undefined. Ideally, one would like the function to behave as represented in Figure 6.3(a) (closely resembling the shape of the distribution of our repository), where being slightly off the optimal case still warrants a high score, but being farther away would warrant progressively lower scores. However, to keep the metric definition as simple as possible, we opted to define the intermediate values by linear interpolation between the aforementioned points, thereby obtaining the following metric definition:

$$SB(n) = \begin{cases} \frac{n-1}{\mu-1} & \text{if } n \leq \mu \\ 1 - \frac{n-\mu}{\omega-\mu} & \text{if } \mu < n < \omega \\ 0 & \text{if } n \geq \omega \end{cases}$$

which behaves as shown in Figure 6.3(b).

The result of this function is thus a number in the range $[0,1]$, where a higher value denotes less deviation from the "optimal number of components", and thus a better decomposition of the system.

6.6.3 Component Size Uniformity

The goal of the Component Size Uniformity (*CSU*) metric is to measure how uniformly the volume of the system is distributed over its components. A way to measure this is by using the Gini coefficient (Gini, 1921; Vasa et al., 2009). The value of the coefficient is a number in the range $[0,1]$, where a low value denotes a more balanced distribution in the population, and a higher value means more inequality, i.e., a small part of the population has a significantly higher value than the rest.

The coefficient is directly applicable to the problem at hand, except that we would like a lower value to represent a less well-distributed decomposition, thus we define *CSU* as:

$$CSU(C) = 1 - Gini(\{volume(c) : c \in C\})$$

The result of this function is a number in the range $[0,1]$, where a higher value denotes a more balanced distribution of the volume of the system into its components.

As discussed in Section 6.4, another option would have been to use the MSUI metric of Sarkar et al. (2007). We evaluate this option and compare it to the *CSU* in Section 6.8.

6.6.4 Component Balance

The values of *SB* and *CSU* need to be combined to give intuitive quantifications to the scenarios listed in Section 6.2. Since there are various ways to combine the two metrics (e.g., minimum, maximum, sum, product, average), we need to choose an aggregation function based on desired properties. The main property we require is that the function is *conjunctive*, i.e., that it does not allow for compensation (Beliakov et al., 2008).

To illustrate, consider the extreme case where a system is decomposed into a single component. This results in a low score on *SB* (0), but a perfect score of 1 on *CSU*. This last score is due to the fact that when a single component holds the complete size of the system, the size is trivially evenly distributed. For example, when using the average (which is a disjunctive function) the resulting score would be $\frac{1+0}{2} = 0.5$, thus assigning a value in the middle of the range for what is considered a very bad component balance.

The simplest examples of conjunctive aggregation functions are the minimum and the product. The minimum, however, reduces the discriminative power of the metric. For example, a system for which $SB = 0.1$ (few components) and $CSU = 0.1$ (badly balanced) would not be distinguishable from another system with the same number of components but for which $CSU = 1$ (perfectly balanced). For that reason we chose the product as the aggregation function, thus:

$$CB(S) = SB(|C|) \times CSU(C)$$

The result of this function is a number in the range [0,1]. Higher values represent better component decomposition.

6.6.5 Properties

The definition of *CB*, as outlined above, exhibits several desirable properties. First of all, the definitions of the metrics are not tailored towards any programming language or methodology, which ensures that requirement **R3** is satisfied.

Secondly, the metrics are not influenced by the volume of the system. As shown before, the number of components is not correlated with the size of the system. Additionally, the Gini-coefficient has specifically been designed to be agnostic to population size. This enables the comparison of the values for a system over time, even when the system grows.

More generally, even though the metric is explained in terms of the components of the software system, it is not necessarily limited to this particular situation. In fact, the definition is generic enough to apply to any situation in which an entity is decomposed into distinct parts with a given size.

Regarding limitations of the metric, we emphasize that it can currently only be applied on *a single level of decomposition*. Therefore the metric cannot be directly used to quantify the analyzability of a multi-layered architecture. However, the metric can be applied to each layer of such a decomposition in isolation, which reduces the impact of this restriction.

6.6.6 Metric instantiation

To instantiate the metric for a specific domain two actions need to be taken. First, to be able to calculate the metric, the free variables of *SB* must be instantiated. In our current situation, we should instantiate μ with the "optimal" number of components for a system.

One line of reasoning here is that the number of components should be such that a developer can efficiently work with them in its short-term memory. Based on this assumption the theories of Miller (1956) would suggest that the number 7 is an appropiate choice. However, this is an hypothesis which, to the best of our knowledge, has not been validated in the field of software engineering.

As a pragmatic alternative, we use "the most common number of components" as an approximation of the ideal number of components. By using the central tendency of the repository described in Section 6.5, we can define $\mu = 8$. The value of ω should be one of the higher values of the metric observed in the repository. However, to not be overly sensitive to extreme values we decided to take the 95th percentile of the repository which is valued at 16. We thus have $\omega = 16$.

Secondly, in most domains a small variation in the size of components is allowed and expected, which means that a value of 1 for *CB* is only achieved under artificial circumstances. Therefore, the distribution of the values of *CB* for real-world cases should be analyzed to determine which values can be considered indicative of a certain level of analyzability, instead of using the theoretical range of the metric.

6.7 Evaluation Design

While the definition of the metric inherently satisfies requirement **R3**, both requirement **R1** and **R2** call for a more extensive evaluation. First of all, we want to evaluate whether *CB* satisfies requirement **R1**, thus we have the following goal:

G1: Evaluate if *CB* can be used as an indicator for the analyzability of a software system.

Whether or not this goal is achieved, it could be the case that taking an alternative choice in the design of the metric would lead to better results, thus it would be interesting to evaluate the alternatives:

G2: Evaluate if the choices made in the design of the metric are appropriate.

In order to achieve these two goals, we designed a quantitative experiment where the metric and some alternatives were compared to a "Gold standard", which consisted of ratings provided by experts. This is presented in Section 6.8.

To satisfy requirement **R2** we need to determine whether *CB* provides relevant results during all stages of the life-cycle of a software system. Thus, we would like to address two goals, namely:

G3: Understand how the value of *CB* can help during the assessment of a software architecture.

G4: Understand how the value of *CB* evolves over time.

To accomplish these goals, we performed a case study where we assess a software system in terms of its structural decomposition. We used the metric as a basis for discussion of the current decomposition, as well as to investigate its evolution through time. This is presented in Section 6.9.

The integrated evaluation findings are presented in Section 6.10, along with an assessment of the threats to validity covered in Section 6.11.

6.8 Quantitative evaluation of metric performance

6.8.1 Experiment design and execution

The first step is to create the "Gold standard" to compare the metric to. To do this, we conducted interviews with eight experts working at SIG (see Section 6.5.2) in the field of software quality assessment, who were asked to rate the analyzability of a given software system in terms of its structural decomposition into components. They were requested to use a 5-point Likert scale, but in certain cases they did not find the scale detailed enough and chose to award half-point ratings.

All experts are experienced in evaluating the technical quality (focussed on maintainability) of industrial systems. For each expert, we selected 1–3 systems for which they had conducted regular, monthly assessments, during at least three months. This time period was chosen to ensure that the expert was familiar with the systems under evaluation. This resulted in 15 different systems of which 10 are implemented in Java, 4 in C# and 1 in VB.NET.

For 6 out of the 15 systems we asked two experts to provide us with an analyzability rating, for the other 9 we could only interview a single expert due to resource constraints. In 4 out of the 6 double analyzability ratings the experts agreed with

System	Language	$\|C\|$	KLOC	Expert Analyzability Rating
A	Java	8	53	2
B	Java	10	153	3
C	VB.NET	2	87	2.25
D	C#	11	22	2
E	C#	9	82	2
F	Java	5	273	3
G	Java	5	64	2.5
H	Java	51	333	1
I	Java	5	35	3.5
J	Java	5	25	3
K	Java	11	145	2
L	Java	14	512	2
M	C#	16	125	2
N	Java	9	197	5

Table 6.1: *Statistics of the systems used in the evaluation*

each other by giving the same analyzability rating. In one case they disagreed only by half a point, so we decided to include the data point using the average of their analyzability ratings (2.25). In another case, one of the experts gave an analyzability rating of 1 while the other expert gave an analyzability rating of 3. Because of this disagreement we excluded those analyzability ratings from the results, resulting in 14 data-points.

The details of the systems used, as well as the analyzability ratings assigned by the experts can be found in Table 6.1. As can be seen, the systems range over different sizes and numbers of components. Furthermore, the experts tend to provide an analyzability rating below the average rating of three, while distributing the analyzability ratings over the full possible range.

6.8.2 Experiment details

In order to ensure that the experts all had the same understanding of the terms used they received an explanation of the overall goal of the Component Balance metric at the start of the interview. It was explained that when the code is evenly distributed over the components on a similar level of abstraction, it is easier to find out where changes in a system need to be made and that, therefore, this metric is related to the analyzability of the system.

To lower the risk that the experts use pre-existing knowledge about metrics related to Component Balance to guess the desired outcome, the experts were made aware of the fact that using this type of knowledge would mean that we would measure their ability to guess a metric. It was stressed that we are only interested in their expert opinion. To emphasize this point and to make the evaluation as realistic as possible, the experts were asked to imagine that the customer behind the system is asking for such an analyzability rating.

To evaluate the technical quality of systems, all experts are experienced users of the SIG Quality Model (Heitlager et al., 2007). This model provides a maintainability rating for a software system on a scale of 1 to 5, in which each level corresponds to 5/30/30/30/5-percent of the systems in a benchmark. In other words, a maintainability rating of 1 puts a system within the worst 5 percent of systems, and assigning a score of 5 places a system amongst the top 5 percent of systems (Baggen et al., 2010; Correia and Visser, 2008). Because the experts are familiar with the usage of such a scale, we have chosen to adopt the same scale for the interviews.

During the analyzability rating-phase, the experts had access to all the data that they usually have while evaluating these systems. This data includes (among other things) size, coupling, complexity and duplication metrics. In addition, dependency graphs between components and the evolution of these dependencies over time were available. In connection to the metric, the experts do have access to the number of components and the different sizes of the components, but they do not have access to the Gini-values for component sizes, nor do they have access to any contextual information provided by a repository such as defined in Section 6.5.

Apart from the actual analyzability rating, the experts were also asked to provide a motivation for it, which was used to a) cross-check analyzability ratings between experts which rated the same systems, b) see whether our initial intuition is shared by the experts, and c) determine which metrics the experts used in their evaluation.

6.8.3 Results

In order to evaluate whether the values of *CB* can serve as an indicator for the opinion of the experts, we calculated a Spearman rank correlation coefficient (ρ) between the analyzability ratings given by them and the Component Balance metric values. This non-parametric rank correlation test was chosen because no assumptions can be made as to how the values extracted from either the experts or the value of *CB* are distributed.

Furthermore, we use the ranking to evaluate some of the choices made in the design of the metric. For example, we combine *SB* and *CSU* by multiplication rather than a different aggregation function. In addition, we have chosen to use the Gini coefficient as a means to calculate *CSU* instead of the *MSUI* metric proposed by Sarkar et al. (2007). To validate whether these decisions are justified, we calculate the Spearman correlation scores for two alternative aggregation functions as well as the replacement of *CSU* with *MSUI*. Lastly, to ensure that combining the two parts of the metric is actually needed we also calculate the correlation scores for *CSU*, *SB* and *MSUI* in isolation. The results are summarized in Table 6.2.

The Spearman rank correlation between the opinion of the experts and the values obtained from *CB* is 0.80 (with a p-value of 6.4×10^{-4}), which indicates a strong, significant correlation between both rankings.

Metric	ρ	p-value
$CB\ (SB \times CSU)$	0.80	0.00064
$CB\ (min(SB, CSU))$	0.79	0.00084
CSU	0.73	0.0031
$CB\ (\frac{SB+CSU}{2})$	0.70	0.0057
$MSUI$	0.68	0.0071
$CB\ (SB \times MSUI))$	0.62	0.019
SB	N/S	0.43

Table 6.2: *Correlations scores between experts ranking and different definitions of CB*

The table shows that almost all of the alternative metrics can be used as an indicator for the opinion of experts, but that the definition for *CB* as given in Section 6.6 yields a significantly higher correlation score than most of the alternative metrics. Only using the minimum as an aggregation function has a similar performance, but as explained in Section 6.6.4, using multiplication is preferable to taking the minimum, because of the added discriminative power.

The only alternative metric for which there is no significant correlation score is using *SB* in isolation. Although there is a reason for not using this metric in isolation, because it focusses on only one aspect of the component structure, the lack of statistically significant evidence does not allow us to reject this metric as a candidate on mathematical grounds. An experiment involving more subjects should be conducted to investigate this hypothesis more thoroughly.

6.9 Case study

6.9.1 Subject system

The subject of the case study is Checkstyle[1], an open-source Java-library that checks for coding violations. This project has been chosen because it is mature (having a history of over 10 years), widely used in both industry and academia, and small enough to be evaluated and understood in a reasonable amount of time. In addition, the open-source nature of the project allows for easy replication.

In this evaluation, we considered the major releases from 1.0 (January 2001) until 5.1 (February 2010). We only considered the Java-code in the main `src` directory of the Checkstyle project, excluding the separate source-tree under `contrib`.

[1] `http://checkstyle.sourceforge.net/`

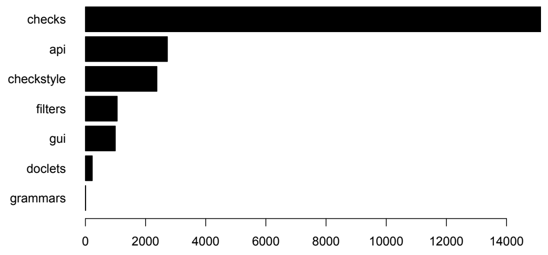


Figure 6.4: *Sizes of the top-level packages of Checkstyle 5.1 measured in Lines of Code.* $SB = 0.86$*,* $CSU = 0.34$ *and* $CB = 0.29$*.*

6.9.2 Architecture assessment

To understand how the value of *CB* can be used in a discussion about the top-level decomposition of a software system, we perform an in-depth evaluation for release 5.1 of Checkstyle, which has a *CB* value of 0.29. By evaluating this value in the context of Checkstyle, we hope to determine actions to be taken to increase the analyzability of the project.

To put the value of 0.29 in perspective, we compare it to the *CB*-values of other systems. In particular, we can compare it to the *CB*-values of the repository established in Section 6.5. It turns out most systems (90%) have a *CB*-value between 0 and 0.53. Checkstyle scores better than about 55% of the systems in the repository, meaning that the quality of its decomposition is slightly above average.

If we breakdown the metric into its two parts, we have $SB = 0.86$ and $CSU = 0.34$. In the context of the repository, the value of *SB* is higher than 84% of the systems, but *CSU* scores in the bottom 22%. This indicates that the system is decomposed into a reasonable number of components, but that there is a large variation in the sizes of those components. The plot of the relative sizes of the components of Checkstyle shown in Figure 6.4 confirms this.

The biggest component is `checks`, containing almost 70% of the code of the system. Given that the core-functionality of the library is to check for coding-standard violations, it is not surprising to notice that most of the code of the project is indeed dedicated to the implementation of the various checks. Given this distribution of size, our intuition is that most of the changes take place within the `checks` component. A manual inspection of the release notes of the project reveals that in the last three releases, most of the features and bug-fixes have indeed been related to different checks.

Based on the fact that checkstyle is a plug-in based architecture, a recommendation could be to split up the project into a “framework”-project and a “plugins”-projects. Benefits of this approach are, amongst others, a more strict separation of concerns on the architecture level and less code to analyze for developers working on checks. In addition, such a break-up would have benefits on the project-management level, possibly resulting in more focussed project-teams and separate release cycles.

In an industry setting, this would also lead to a new allocation of budget and other resources.

To evaluate the result of such a break-up we calculated the values for *CB* for both the hypothetical "framework" project (all top-level packages minus the `checks` packages) and the hypothetical "plugins" project (the top-level `checks` package which is decomposed into sub-packages). For the framework project, the value of *CB* would rise to 0.39, placing it in the higher regions of our repository. In the other hand, the score for the plugins project would be valued at $CB = 0$. This low score is due to the fact that $SB = 0$, which in turn is caused by the fact that the `checks`-package is subdivided into 16 different sub-packages.

A closer inspection of the naming of these sub-packages reveals that not all of them are on the same level of abstraction. There are both sub-packages with generic names such as `design`, `coding` and `metrics`, as well as sub-packages with a more specific purpose such as `modifier`, `header` and `annotation`. This last sub-package, together with the sub-package `blocks`, could probably be merged into the more generic sub-package `code`. In addition, the grouping of the checks in the different sub-packages is not consistent. For example, there is a sub-package called `whitespace`, but the `NewlineAtEndOfFileCheck` is placed in the root of the `checks`-package, right along abstract types for checking formatting and options. Again an example of placing concepts from different levels of abstraction side-by-side.

These observations strengthen the evaluation by *CB* that within this part of checkstyle it is indeed hard to determine where certain functionality is located. A reorganization of the checks into clearly defined concepts on a similar level of abstraction would help new developers to understand more easily where to look for specific checks.

6.9.3 System evolution

To understand how the value of *CB* evolves over time we plotted the values of *CB*, *SB* and *CSU* for 23 releases of Checkstyle in Figure 6.5. In this plot, four distinct time periods (marked as *I*, *II*, *III* and *IV*) can be distinguished. For each of these periods, we relate the changes in the values of the metrics to specific events in the development of the project and their impact on the analyzability.

In the first time period *I*, corresponding to the releases 1.0 to 3.0, one can observe that the metric is valued at 0. In fact, initially Checkstyle had no major component decomposition. At the end of the period, we see a large positive jump. The explanation for this jump can be found in the release notes of release 3.0, which state: *"Completely new architecture based around pluggable modules."* This release decomposed the system into three distinct components; the core, the API and the checks, thus making it easier to distinguish between these three types of functionality.

The second time period *II* shows a sharp decrease in *CB*-value between releases

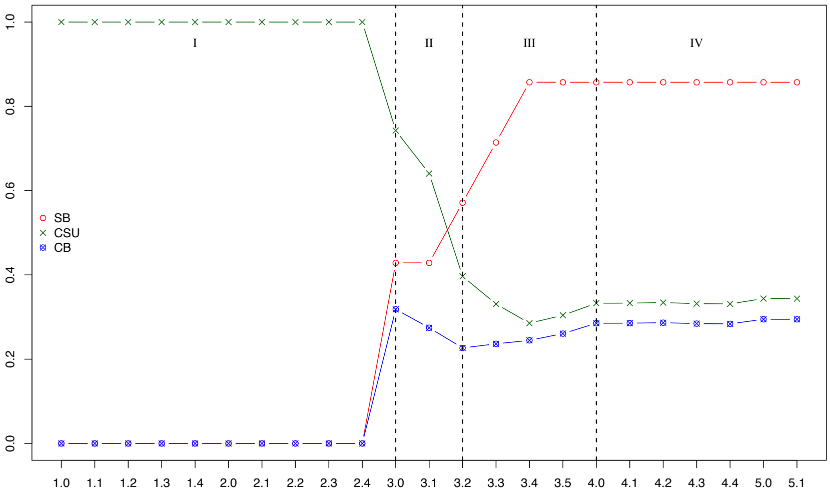


Figure 6.5: *Metric Trends for different Checkstyle versions*

3.0 and 3.2. This decrease is mostly due to the sharp decline in the values of *CSU* which suggest that a large amount of code was added to a single component. An inspection shows that this is indeed the case, since the largest component `checks` almost doubled in size between both releases 3.0 and 3.1 and between release 3.1 and 3.2, while the size of the other components changed only a little.

Within the third time period *III*, spanning releases 3.2 until 4.0, the value of *CB* slowly increases. In the first two releases of this period, this is due to the increasing value of *SB*, which is explained by the addition of the components `grammar` and `doclets`. In the later two releases, the number of components is stable, thus the increase in value can only be explained by an increase in *CSU*. Inspection of the code shows that between these releases several checks have been retired to a separate source tree which results in almost a 20 percent decrease of code in the `checks` component. Although removing this code requires a software engineer to examine less code in the largest component, the overall analyzability only increases a little.

Since release 4.0, the value of *CB* has not been changing significantly, indicating the architecture of the project has stabilized in time period *IV*. Inspecting the release notes confirms this, since after release 4.0 no mention to architectural changes on the level of components can be found.

6.10 Discussion

This section discusses the results of the two phases of the evaluation with respect to the goals as outlines in Section 6.7.

In Section 6.8.3 we showed that the value of *CB* strongly correlates with a ranking of the analyzability of a software system as given by experts. Because of this, we conclude that *CB* can be used as a proxy for the analyzability of a system and thus **G1** is satisfied. In addition, the results also show that alternative implementations of the metric do not outperform the definition as given in Section 6.6, thus satisfying **G2**.

In order to evaluate *why CB* shows this strong correlation, we took a closer look at the systems which where assigned similar rankings by both the experts and *CB*. The system which is ranked lowest by both the *CB*-metric and the experts consists of over 50 components which is the result of combining both a technical and functional decomposition on a single level. This results in components which are either dedicated to the implementation of a single business functionality, or contain the complete GUI of the system. On the high end of the scale there is the system which is ranked the highest by both the experts, as well as the metric. According to the expert who evaluated this system, the naming of the components suggest a similar level of abstraction.

There are two cases in which *CB* gave a significantly higher ranking than the experts. One of these systems was decomposed into eight components, of which two contained almost all of the code. According to the experts, the architecture aims to be a Model-View-Controller architecture, in which the representation of the model of the application is separated from the implementation of the manipulation and display functionalities. However, in the implementation of this system, both the model and the functionality to manipulate this model are placed in a single component, which leads to a highly skewed implementation. The situation was similar for the other system which has nine components, two of them containing too much functionality. In both cases, the score for *CSU* is low, but the high score for *SB* places the systems on a higher position than the expert(s).

For none of the systems the ranking of *CB* was significantly higher than the ranking given by the experts. Given the previous example, it could be the case that a high score for *SB* has a too strong influence on the value of *CB*. This might be because reaching an optimal value for *SB* is relatively easy, while reaching an optimal value for *CSU* is very difficult under realistic scenarios. To compensate for this, we could make the optimal value for *SB* harder to reach or choose a more complex aggregation function. Both solutions might lead to better results at the cost of a more complicated definition of the metric, which is undesirable from an explainability point of view. Before exploring alternative options we first assess the impact of the stronger influence of *SB* when using *CB* in practice in Chapter 9.

With respect to both **G3** and **G4** several observations can be made. First of all, the case-study shows that the metric provides a solid quantified basis for discussing the decomposition of the subject system. In addition, the values of *CB* were show to indicate problems in the code which are related to the analyzability of the system.

As explained in Chapter 5 a metric should always be used in combination with other metrics to come to a well-balanced conclusion. In our architecture evaluation checklist (see Chapter 4), the metric is used to answer only one of 17 questions related to the structural decomposition, thus ensuring that other aspects are also taken into account. If used outside this context *CB* should be complemented with metrics related to the dependencies between the components of a system. We investigate which coupling and cohesion metrics best complement *CB* in Chapter 8.

As described in Section 6.6, the definition of the metric is currently limited to a single level of decomposition. Nevertheless, the metric could be extended to take into account hierarchical decompositions by using the inverse of the *CB* of each component as weights for the Gini coefficient on a higher level. The reasoning behind this is that the influence of the size of a component to the overall inequality in size distribution should be lower the better it is itself properly decomposed into sub-components. Again, before exploring alternative options we first assess the impact of this limitation by applying *CB* in practice in Chapter 9.

6.11 Threats to Validity

To put the results presented in Section 6.10 in perspective, here we address several questions related to the evaluation design.

A first threat that needs to be addressed is the presence and influence of ties in the expert ratings in Section 6.8. To evaluate its impact, we compared the tied expert ratings and the metric rankings. This revealed that, for the two groups of ties, the corresponding *CB* ranks are contiguous with the exception of two cases. To quantify the impact of these ties we examined two scenarios of breaking up the ties. The first one corresponds to a worst-case scenario in which the opinion of the experts is the opposite of the ranking of *CB*, while the second scenario illustrates the best case in which the expert opinion gives the same ranking. This results in a significant correlation score between 0.6 for the worst case, and 0.9 in the best case scenario. Since both scenarios show a strong correlation the influence of these ties is considered limited and does not invalidate our conclusions.

An important question related to the scope of the results is how far the results of the validation can be generalized (external validity). The projects used in the quantitative evaluation are all industry systems written in modern object-oriented languages, but differ in size, application domain and number of components. In principle, the generalizability of the results is limited to these types of systems. Nevertheless, since the metric is not specific for any language nor has any preference towards industry or open-source systems this generalization is possible. To more formally validate this claim we apply the metric to a more diverse set of systems in Chapter 9.

Apart from the set of systems used in the evaluation, the generalizability of the results is also limited by the choices made in the instantiation of the repository (Section 6.5.2). For example, using legacy systems, rather than modern ones, might lead to a different instantiation of *SB*, and thus lead to different results. A preliminary examination of our metric for some COBOL and Pascal systems revealed that they do not stand out as outliers with respect to the systems in our repository. However, more empirical work is needed to determine the impact of taking into account legacy systems. Chapter 9 provides the first steps towards such a validation.

Concerning the repository, we only examined software systems from the point of view of their main technology. However, today's software systems usually consist of multiple languages. To evaluate these hybrid systems one can either evaluate the decomposition of the system per language, or evaluate the system by combining all the languages. The latter option seems to be preferred as it provides a more accurate overview of the decomposition of the system as a whole.

Lastly, a fact that influences the generalizability of the results is the use of industry experts working at a single company. To evaluate whether the same results can be obtained with different experts, the experiment described in Section 6.8 should be replicated using a more heterogeneous set of experts. However, the exact replication of the study is not possible due to the confidential nature of the industry systems used. This threat could have been countered by asking the experts to evaluate freely available open-source systems, but that would reduce the value of their opinion due to less familiarity with the systems. To counter the reduced repeatability, we have explicitly chosen an open-source system as the subject of the case-study and mixed industry and open-source systems in the repository used to instantiate the metric.

6.12 Conclusions

In this chapter, we make the following contributions:

- We describe an empirical exploration of how systems are decomposed into top-level components;
- We define a metric to measure the balance of components which is usable across all life-cycle phases of a project;
- We show how the metric is correlated with the opinion of experts about the analyzability of a software system.

Although the metric was developed in the context of the evaluation of software architecture, its definition is not constrained to a particular language, programming paradigm or level of abstraction. Actually, the metric can in theory be applied to any entity which is decomposed into a discrete set of components with a given size. This allows for the application of the metric on the requirements documentation of a software system, or to measure the quality of the section breakdown of a scientific publication. Although interesting, the investigation of these applications of the metric are considered to be out of scope for this thesis.

CHAPTER 7

Dependency Profiles for Software Architecture Evaluations [*]

Abstract

In this chapter we introduce the concept of a "dependency profile", a system level metric aimed at quantifying the level of encapsulation and independence within a system. We verify that these profiles are suitable to be used in an evaluation context by inspecting the dependency profiles for a repository of almost 100 systems. Furthermore we outline the steps we are taking to validate the usefulness and applicability of the proposed profiles.

7.1 Introduction

Software architecture is loosely defined as "the organizational structure of a software system including components, connections, constraints, and rationale" (Kogut and Clements, 1994). Since the architecture of a software system greatly influences all of a system's quality attributes (Clements et al., 2002), it is important to regularly evaluate the actual, as-implemented, software architecture of a system.

In order to reduce the amount of time and effort needed to perform such an evaluation, an evaluator can use software metrics to spot outliers and identify areas within a system which are in need of a more detailed evaluation. Additionally, the use of metrics reduces the need for expert opinion, thus making the evaluation more objective and repeatable.

[*] Originally published in the proceedings of the 27th IEEE International Conference on Software Maintenance (ICSM 2011) (Bouwers et al., 2011c).

For a metric to be useful in an evaluation context, several characteristics are desirable (Heitlager et al., 2007). For instance, the metric needs to be *simple to explain* to ensure that non-technical decision makers can understand them. Furthermore, in order to allow an evaluation of a diverse application portfolio the metrics should be *as independent of technology as possible*. The ability to perform a *root-cause analysis* is also desirable to ensure that the metrics can provide a basis to determine which actions need to be taken. Lastly, metrics which are *easy to implement and compute* are desired as to reduce the initial investment for performing evaluations.

Research on metrics for software architectures has traditionally focussed on the way components depend on each other and how components are internally structured (coupling and cohesion (Stevens et al., 1974; Yourdon and Constantine, 1979)). To the best of our knowledge, all of the existing metrics for architecture level dependencies fail to meet at least one of the desired characteristics outlined above.

In this chapter we propose the concept of a *dependency profile* which categorizes all modules in a system based on their dependencies. This purpose of the dependency profile is two-fold. On one hand it is aimed at capturing the degree in which the components within a system encapsulate the functionality they offer. On the other hand, the profile quantifies the degree to which components depend on each other. We assess to what extent the dependency profile meets the four criteria discussed above by examining a benchmark of almost 100 systems totaling over 12.5 million lines of code. Additionally we outline a plan to validate the profile against the type of changes that occur within a system.

7.2 Background

To illustrate why existing metrics for quantifying the dependencies between components of a system are less suitable to be used in an evaluation context we present a short overview of typically found shortcomings.

To start, metrics which are simple to explain such as the basic number of incoming and outgoing dependencies allow for root-cause analyses. However, since larger systems tend to have a higher number of dependencies these metrics should be normalized against the size of the system to allow systems of various sizes to be compared.

More complex coupling/cohesion metrics such as those defined by Briand et al. (1999a) or in the well-known C&K suite of metrics (Chidamber and Kemerer, 1994) (including variations), suffer from the same problem of not being normalized against the size of the software unit they are measuring. Additionally, these class-level metrics are designed to target systems written in object-oriented languages, while ideally a metric would be independent of technology.

And although there are extensions to these coupling metrics that are normalized, see for example Gui and Scott (2007), the proposed normalization process tends to decrease the ability to perform root-cause analyses because the outliers in the data, which are the interesting data-points, are usually hidden by the normalization. The same problem applies to metrics defined to rank cluster algorithms, for example the Modularization Quality-metric defined by Mancoridis et al. (1999).

7.3 Dependency Profiles

We define a metric to quantify the dependencies within a system by placing all *modules* of a system (e.g., Java classes or C files) into four distinct categories. This categorization is based on the way in which the modules are grouped into *components* (e.g., Java packages or C directories) and how the modules interact with modules outside their own component.

7.3.1 Terminology

Let $S = \langle M, C, D \rangle$ be a system, consisting of a set of modules M, a set of components C and a set of dependencies between modules D. Each module is assigned to a component and none of the components overlap. More formally, the set $C \subseteq \mathcal{P}(M)$ is a partition of M, i.e.,

- $\forall c_1, c_2 \in C : c_1 \neq c_2 \Rightarrow c_1 \cap c_2 = \emptyset$.
- $\bigcup_{c \in C} = M$

For $(m, m') \in D$ we write $m \rightarrow m'$ to represents a directed dependency from module $m \in M$ to module $m' \in M$.

For each module $m \in M$ it is possible to obtain the containing component through a function $component : M \rightarrow C$. In addition, for a component $c \in C$ we use $\overline{c}$ to denote the complement of c, i.e., all modules not contained in c.

Lastly, each module has a given size (for example measured by the lines of code), which is captured by a function $size : M \rightarrow \mathbb{N}$. The volume of a component $c \in C$ is defined simply as the sum of the size of its modules, thus:

$$volume(c) = \sum_{m \in c} size(m)$$

7.3.2 Types of code

Each module within the components of a system can be divided into one of four categories, see Figure 7.1:

- *Hidden modules* (1): modules which only have dependencies (either incoming or outgoing) involving modules inside the component.

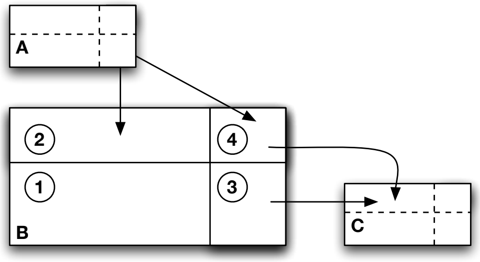


Figure 7.1: *Three components illustrating the four different types of modules within a system; 1) hidden modules, 2) inbound modules, 3) outbound modules and 4) transit modules. Arrows denote dependencies from/to modules within other components.*

- *Inbound modules* (2): modules which do not have outgoing dependencies to modules outside the component, but have incoming dependencies from modules outside the component.
- *Outbound modules* (3): modules which do not have incoming dependencies from modules outside the component, but have outgoing dependencies to modules outside the component.
- *Transit modules* (4): modules which have dependencies (both incoming and outgoing) coming from/going to modules outside the component.

For each of these categories a function of type $C \to 2^M$ can be defined which, given a component C, returns the set of modules within that component which belong to that category. Table 7.1 lists the definitions of those functions. Using these functions, each category of modules can be turned into a normalized metric by calculating the percentage of code in a system which belongs to each category. For example, the percentage of *hiddenCode* of a system is defined as:

$$hiddenCode(S) = \sum_{c \in S} \frac{volume(hiddenModules(c))}{volume(c)}$$

Definitions of the metrics for *inboundCode*, *outboundCode* and *transitCode* are similar.

7.3.3 Dependency Profile

Using the metrics defined above we define a *Dependency Profile* as a quadruple of the four types of code:

$$\begin{array}{rl} \langle \;\; hiddenCode(S) & , inboundCode(S) \\ outboundCode(S) & , transitCode(S) \;\; \rangle \end{array}$$

Name	Collection
$\mathit{hiddenModules}(c)$	$\{m \in c \mid \nexists\, m_i \in \bar{c} : m_i \rightarrow m \in D \wedge \nexists\, m_o \in \bar{c} : m \rightarrow m_o \in D\}$
$\mathit{inboundModules}(c)$	$\{m \in c \mid \exists\, m_i \in \bar{c} : m_i \rightarrow m \in D \wedge \nexists\, m_o \in \bar{c} : m \rightarrow m_o \in D\}$
$\mathit{outboundModules}(c)$	$\{m \in c \mid \nexists\, m_i \in \bar{c} : m_i \rightarrow m \in D \wedge \exists\, m_o \in \bar{c} : m \rightarrow m_o \in D\}$
$\mathit{transitModules}(c)$	$\{m \in c \mid \exists\, m_i \in \bar{c} : m_i \rightarrow m \in D \wedge \exists\, m_o \in \bar{c} : m \rightarrow m_o \in D\}$

Table 7.1: *Conditions for each of the four categories of modules*

A typical instantiation of such a profile is $\langle 75\%, 10\%, 15\%, 5\% \rangle$, which means that 75 percent of the volume of the system falls into the *hiddenCode*-category, 10 percent falls into the *inboundCode*-category, etc. We hypothesize that this dependency profile can be used to quantify two quality aspects of a software system: the degree of encapsulation and the degree of independence.

The concept of *encapsulation* is often used to refer to the level in which the implementation details of functionality are abstracted away by an interface. A high level of encapsulation is desirable since this should mean that changes to the implementation can be done without the need to change clients which are using the interface. We expect that the *inboundCode* metric can be used to measure this quality aspect. To illustrate we compare a system A with a dependency profile of $\langle 50\%, 30\%, 18\%, 2\% \rangle$ with a system B with a dependency profile of $\langle 50\%, 15\%, 34\%, 1\% \rangle$. In system A there is a higher percentage of code which is called from outside the component in which it is defined, which leads to a higher chance that a change in this specific component propagates to other components in the system. We hypothesize that a high value of *inboundCode* shows that there is a low level of encapsulation in the system.

Analogously, *independence* is used to refer to the level in which components of a system rely on other components (either interface or implementation) in the implementation of their own functionality. A high level of independence is desirable since this should mean that changes in modules outside the component should not propagate to the component itself. We expect that the *outboundCode* metric can be used to measure this quality aspect since this metric quantifies the portion of the system which is used by other components. In the example systems above, system B has a higher percentage of code which depends on code outside the component in which it is defined. This leads to a higher chance that a change in a component will propagate to this specific component. We hypothesize that a high value of *outboundCode* indicates that there is a low level of independence in the system.

In both cases the percentage of *transitCode* should also be taken into account. This category contains those modules which both use and are used by modules in other components and are thus even more likely to propagate changes between components. Because of this issue, we hypothesize that although there might be some need for *transitCode*, for example in a component which connects two other components, it is desirable to have a low percentage of *transitCode* in a system.

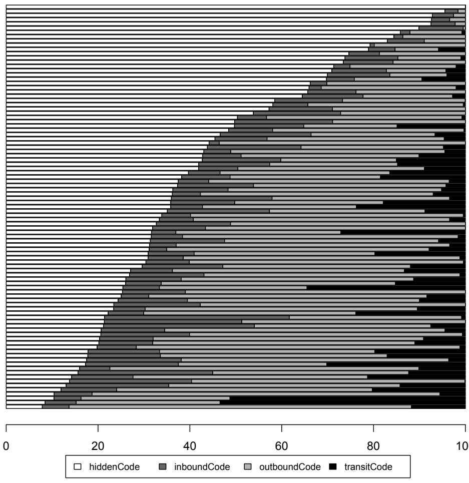


Figure 7.2: *Dependency profiles for a repository of systems, ordered by the percentage of hiddenCode. Each line represents a system.*

7.4 Preliminary Observations

As a first evaluation of the dependency profiles we instantiate the above metric framework and use a repository of systems to observe the distribution for this specific instantiation. The repository is an extended version of the one used in Chapter 6 and contains systems of different sizes, development context (open-source versus industry) and technologies. The following table characterizes the repository in terms of number of systems per technology and development context:

	Java	**.NET**	**C/C++**	**Total**
Industry	45	17	6	68
Open source	20	4	3	27
Total	65	21	9	95

To ensure that the metrics can be calculated for all technologies we instantiate “module” as a source-file, “dependency” as a direct call relation and “component” as the first level of decomposition in the system. Determining the components of the systems follows the approach in Chapter 6, i.e., for all systems the top-level decomposition was made by a technical analyst of the Software Improvement Group based on the directory structure of the system and available documentation. For the industry systems this decomposition was validated with the development team. A chart showing the distribution of the dependency profiles for this repository and this instantiation is given in Figure 7.2.

7.4.1 General Observations

A first observation that is clear from Figure 7.2 is that the percentage of *hiddenCode* differs considerably for the systems in the repository, ranging from 7 to 100 percent with a median of 35 percent. Since having 100 percent of *hiddenCode* is strange, we investigated this particular system, an industry system written in C#, and found that each top-level component in the system was a specific service built upon an external framework. Since each service is independent from the all other services none of the services have code in common.

Another observation that can be made about the distribution is that a large portion of the repository (18 systems) does not have any *transitCode*, which corresponds with our initial expectation. However, the amount of *transitCode* rises to over 20 percent for 10 systems, having a maximum of 53 percent in the repository. Within these systems we expect to see a high frequency of propagating changes. In Section 7.6 we provide an outline of how we plan to validate this hypothesis.

A last observation that can be made is that in almost all cases (only 9 exceptions) the amount of *outboundCode* is larger than the amount of *inboundCode*. This could indicate that, in general, there is a stronger focus on the design of the *provided* interface of a component than on restricting the *required* interface of a component.

7.4.2 Statistical Observations

To enable a fair comparison between systems of different sizes we need to ensure that there is no strong correlation between the size of the systems and the percentages in the dependency profiles. To assess this we use a Spearman rank correlation test using the size of the system in lines of code and the percentage of code in each of the four categories. No significant correlations were found for *hiddenCode* and *transitCode*, while both *inboundCode* and *outboundCode* have a weak correlation of -0.28 and 0.32 ($p < 0.01$) respectively. Thus we can conclude that there is no strong correlation between the size of a system and any of the four categories.

In addition, using a two-sided Kolmogorov−Smirnov test we can determine whether there is a significant difference between the distributions of two data samples. Using this test we did not find any significant differences between the distribution of the values for different development contexts (industry versus open-source) or system type (application versus libraries). However, there are differences between the distribution in the values of *hiddenCode* for the Java technology versus other technologies. Inspecting the distributions shows that systems written in Java tend to have a lower percentage of *hiddenCode*. This difference in distribution does not mean that the metrics are technology dependent, but only that the metrics might consistently produce lower values for certain technologies.

7.5 Discussion

As discussed before there are four desirable characteristics of metrics to be useful in an practical evaluation setting (Heitlager et al., 2007). We argue that the metrics used in the dependency profile as described in Section 7.3 feature these characteristics.

First of all, the metrics should be *simple to explain*. Even though the formal definition of the metrics can be considered complex the intuition behind the metrics is easy to explain given the visual support of Figure 7.1.

Secondly, the metrics should be *as technology independent as possible*. The definition of the four metrics contains no technology specific constraints, although certain definitions of “module” or “dependency” could make the metrics technology specific. By using a generic instantiation of the metrics as given in Section 7.3 there are no practical problems in comparing systems written in different technologies.

Furthermore, the metrics should allow for a *root-cause analysis*, which is relatively straight-forward. After first using the system level dependency profiles to discover a system in need of further investigation, the profile can be calculated on component level to determine which component contributes the most to each category of code. After the most interesting component has been found, the modules in the category of interest can be sorted according to their size to discover which module is contributing the most to this category. After determining the most interesting modules an expert should inspect the dependencies to/from these modules to determine *why* these dependencies are there and whether they are problematic.

Lastly, the metrics should be *straight-forward to implement*. Because the metrics are based on basic data such as dependencies between modules and the size of a module, any existing tool which is capable of calculating this data should be able to implement the needed metrics with only a small amount of effort.

7.6 Evaluation Design

To determine whether the intuition as described in Section 7.3.3 is correct we plan to test the following hypothesis in the remainder of this thesis:

- Systems with a low percentage of *inboundCode* plus *transitCode* (e.g., a small *provided interface*) have a better encapsulation and therefore changes in a component will less likely propagate to other components

- Systems with a low percentage of *outboundCode* plus *transitCode* (e.g., a small *required interface*) have more independent components and therefore changes in a component will less likely propagate to other components

To validate these hypothesis we perform a case-study in which we examine the change-sets of a system using the framework proposed by Yu et al. (2010). This framework defines co-evolution of a system as either being internal (i.e., all modules in a change-set belong to a single component) or external (a change-set contains modules of multiple components).

As a first step we calculate the frequency of external co-evolutions for a number of open-source systems. By correlating this frequency with the size of the provided interface, the size of the required interface, and a combination of these two measures we plan to validate or reject the two hypothesis above. The details of the design and result of this study are described in Chapter 8.

In addition, a more qualitative study will be performed in which (part of) the profiles are embedded within a measurement model for quantifying the maintainability of a software system. The goal of this study is to determine whether the dependency profiles are useful in an evaluation setting. The details of the design and the results of this qualitative study are described in Chapter 9.

7.7 Conclusions

This chapter makes the following contributions:

- The definition of a dependency profile with desirable characteristics for use in a software evaluation setting
- A first analysis of these profiles using a large repository of systems
- An outline of the evaluation strategy for the profiles

The results of setting up and performing the evaluation experiment are presented in Chapter 8.

CHAPTER 8

Quantifying the Encapsulation of Implemented Software Architectures *

Abstract

In the evaluation of implemented software architectures, metrics can be used to provide an indication of the degree of encapsulation within a system and to serve as a basis for an informed discussion about how well-suited the system is for expected changes. Current literature shows that over 40 different architecture-level metrics are available to quantify encapsulation, but empirical validation of these metrics against changes in a system is not available.

In this chapter we survey existing architecture metrics for their suitability to be used in a late software evaluation context. For 12 metrics that were found suitable we correlate the values of the metric, which are calculated on a single point in time, against the ratio of local change over time using the history of 10 open-source systems. In the design of our experiment we ensure that the value of the existing metrics are representative for the time period which is analyzed. Our study shows that one of the suitable architecture metrics can be considered a valid indicator for the degree of encapsulation of systems. We discuss the implications of these findings both for the research into architecture-level metrics and for software architecture evaluations in industry.

[*] Submitted to ACM Transactions on Software Engineering and Methodology on September 6th, 2012.

8.1 Introduction

When applied correctly, the process of encapsulation ensures that the design decisions that are likely to change are localized (Booch, 1994). In the context of software architecture, which is loosely defined as "the organizational structure of a software system including components, connections, constraints, and rationale" (Kogut and Clements, 1994), the encapsulation process revolves around hiding the implementation details of a specific component. Whether the encapsulation was done effectively, i.e., to what extent changes made to the system were indeed localized, can only be determined retrospectively by examining the history of the project.

However, for evaluation purposes it is desirable to use the current implementation or design of a system to reason about quality attributes such as the encapsulation of the system (Clements et al., 2002). A common approach for this is to use a metric which can be calculated on a given snapshot of the system, i.e., the state of a system on a given moment in time, to reason about changes that will occur subsequently.

On the class level, several metrics for encapsulation have been proposed, see for example the overview by Briand et al. (1999a), which have also been evaluated empirically through a comparison with historic changes (Lu et al., 2012). At the broader system level, however, few metrics for encapsulation exist, and for those that exist no empirical validation against historic changes has been provided (Koziolek, 2011). The goal of this chapter is to fill this gap by means of an empirical study.

In our empirical study we take into account system-level architecture metrics capable of quantifying the encapsulation of a system during the evaluation of an implemented software architecture. These so-called "late" architecture evaluations (Dobrica and Niemelä, 2002) are conducted to determine which actions need to be taken to improve the quality of an architecture when, for example, the current implementation deviates substantially from the original design or when no design documents are available. Moreover, performing this type of architecture evaluation on a regular basis helps in identifying and preventing architecture erosion (Perry and Wolf, 1992). Additionally, this type of analysis is also employed to get a first overview of the current state of a large portfolio of systems.

In such a context, the relative ease of calculation of the metrics is important to ensure that (repeated) evaluation is economically feasible. In addition, the ability to perform root-cause analysis is a key-factor in deriving low-level corrective actions based on the high-level metrics, which in turn helps in the acceptance of the metrics with practitioners (Heitlager et al., 2007). Moreover, metrics which can be calculated on a broad range of technologies are preferred in order to make it possible to compare systems taken from a large, heterogeneous portfolio of systems. The first step in our study is to identify a set of existing metrics that adhere to these properties, which is done by surveying the over 40 available architecture-level metrics.

In the second step of the study the remaining 12 architecture-level metrics are correlated with the effectiveness of encapsulation in 10 open-source Java systems having an average history of six years. To quantify historic encapsulation, we follow the proposal of Yu et al. (2010) and classify each change-set in a system as either *local* (all changes occur within a single component) or *non-local* (multiple components are involved in the change). A high ratio of local change-sets shows that the frequently changing parts of the system were indeed localized, indicating that the encapsulation was done effectively.

The results of our study show that three of the suitable architecture metrics, those that are aimed at quantifying the extent to which components are connected to each other, are correlated with the ratio of local change-sets. Of these three, one is chosen as a valid indicator for the effectiveness of encapsulation of a system. In contrast, metrics which are purely based on the number of components or on the number of dependencies between components were not found to bear a relationship to the effectiveness of encapsulation. The implications of this finding on both the research into architecture-level metrics as well as the use of these metrics in an architecture evaluation setting are discussed.

8.2 Problem Statement

Following the Goal Question Metric approach of Basili et al. (1994) we define the *goal* of our study to be to evaluate existing software architecture metrics *for the purpose of* assessing their indicative power for the level of encapsulation of a software architecture. The *context* of our study is late software architecture evaluations *from the point of view* of software analysts and software quality evaluators. From this goal the following research question is derived:

> *Which software architecture metrics can serve as indicators for the effectiveness of encapsulation of an implemented software architecture?*

In order to answer our research question we first survey the currently available architecture metrics for their suitability to be used in a late software architecture evaluation context in Section 8.3. We then design and execute an empirical study to determine the relationship between the selected software architecture metrics and the effectiveness of encapsulation. The design and implementation of this study is given in Section 8.4 and Section 8.5, the results are presented in Section 8.6. In Section 8.7 the presented results are discussed and put into context. Section 8.8 discussed threats to validity, after which related work is discussed in Section 8.9. Lastly, Section 8.10 concludes the chapter.

8.3 Metrics for Encapsulation

As a first step towards answering our research question we review the architecture metrics currently available in the literature. The purpose of this review is to select

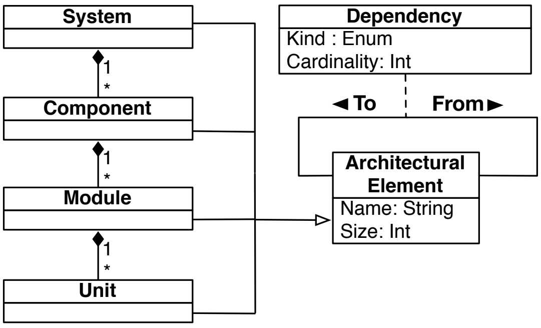


Figure 8.1: *UML-diagram of the used architectural meta-model*

those metrics which are a) capable of providing a system-level indication of the encapsulation of a software system, and b) can be used within a late architecture evaluation context. Before discussing the selection criteria we first introduce the model we use to reason about an implemented software architecture.

8.3.1 Architectural Meta-Model

In this chapter, we look at the software architecture from the module viewpoint (Clements et al., 2003). The model we use is displayed in Figure 8.1 in the form of a UML-diagram. We define a *system* to consist of a set of high-level *components* (e.g., top-level packages or directories) which represent coherent chunks of functionality. Each component contains one or more *modules* (e.g., source files). Within modules, a *unit* represents the smallest block of code which can be called in isolation. Each module is assigned to a single component and none of the components overlap.

Directed dependencies exists between both modules (e.g., extends or implements relations) and units (e.g., call-relations) and have an attribute cardinality which represent the number of dependencies between two units. Dependencies between components can be calculated by lifting the dependencies from the modules/units to the component-level. Both modules and units can have (code-based) metrics assigned to them, for example lines of code or McCabe complexity (McCabe, 1976), which can be aggregated from the module-/unit-level to the component-/system-level.

8.3.2 Metric Selection Criteria

We use the criteria as listed in Table 8.1 to identify metrics that are relevant to our experiment. The first two criteria expresses our focus on system level architecture metrics capable of quantifying the encapsulation of a system. Criteria C3, C4 and C5, which have been identified by Heitlager et al. (2007), relate to the suitability of a metric for use within a late architecture evaluation context. Lastly, criterion C6 ensures that a metric can be used to compare different snapshots of a system over

	The architecture metric:
C1	has the potential to measure the level of encapsulation within a system
C2	is defined at (or can be lifted to) the system level
C3	is easy to compute and implement
C4	is as independent of technology as possible
C5	allows for root-cause analysis
C6	is not influenced by the volume of the system under evaluation

Table 8.1: *Criteria used for the selection of architecture metrics*

time. Together, these six criteria ensure that the metrics can be used to evaluate the encapsulation of a system in a repeated manner.

To ensure criterion C1 (measuring encapsulation) we follow the definition of Booch (1994) and equate the level of encapsulation with the locality of changes. Using this definition we analytically assess each metric to determine to what extent the metric quantifies the propagation of changes from one module to another. However, metrics which do not have a (straight-forward) potential for measuring the level of encapsulation within a system are not automatically excluded from our experiment. Instead, we include these metrics as control variables in order to verify the validity of our experimental set-up.

To ensure criterion C2 (system level metric) we check whether a metric defined at the component level can also be calculated on the level of a complete system. If this is the case (for example by using the dependencies on a higher level) we include the metric in our experiment. If the metric can only be calculated on the level of a component the metric is excluded. Determining the best way to aggregate component level metrics to the system level (i.e., determining the best aggregation function for each metric) is considered to be a separate study. This criterion ensures that the metric can be used to compare different systems.

To ensure criterion C3 (that the metric is easy to compute), we adopt an architectural description making as little assumptions as possible. Following the terminology as defined in Section 8.3.1 this minimal description consists of a) the source-code modules of a system, b) the dependencies between these modules (e.g., call relations), and c) a mapping of the source-code modules to high-level components. For most programming languages, tool support exists to identify the required architectural elements and their dependencies, thus making metrics based on these inputs easy to implement which reduces the initial investment for performing evaluations.

To ensure criterion C4 (technology independence) we restrict the module-level metrics to those that can be applied to any technology. Consequently, metrics that are specific to, e.g., object-oriented systems (such as the depth-of-inheritance tree) are not taken into account in the current experiment. This criterion enables the evaluation of a diverse application portfolio.

To ensure criterion C5 (allow for root-cause analysis) we require that it is possible to identify architectural elements that cause an undesirable metric result. In other words, when the final value of the metric is undesired (i.e., either too high or too low) it should be possible to determine which source-code modules are the cause of the value (and are thus candidates for further inspection). This criterion ensures that the metrics can provide a basis to determine which actions need to be taken to improve the system if needed.

Heitlager et al. (2007) also identified a fourth criterion, that a metric is simple to explain to ensure that technical decision makers can understand them. Because we have no objective way to determine whether a metric is simple to explain this criterion is not used in the selection process.

To ensure criterion C6 we do not include metrics that are influenced by the volume of the system under review in a straight-forward manner (e.g., the absolute number of calls between components) in the current experiment. This criterion ensures that the selected metrics can be used to compare systems with varying sizes, thus allowing the comparison of different systems across a portfolio or the comparison of different versions of the same systems over time. Determining the best way to normalize this type of metrics is outside the scope of the current research. Note that the number of components is independent of the size of a system. The empirical study presented in Chapter 6 established that the number of top level components of any system (irrespective of the volume) revolves around 7.

8.3.3 Metric Selection

Using the overview of Koziolek (2011) as a basis we identified over 40 metrics in the literature, of which we provide a full account in Section 8.11. In the remainder of this section we discuss the most important ones, including the 12 which are included our in experiment. Of these 12, nine have a clear conceptual relation to encapsulation while three metrics are included as control variables.

Briand et al. (1993) define three different coupling and cohesion metrics. First, three definitions of the *ratio of cohesive interactions* (RCI) are given. As input for these ratios the concepts of “known”, “possible” and “unknown” interactions need to be instantiated. In order to make it possible to calculate these metrics on a large scale we consider all dependencies between source-code modules to be “known” interactions. Furthermore, following the definition of the pessimistic variant of the metric all “unknown” interactions are treated as if they are known not to be interactions. Then, we define the RCI to be the division of the number of “known” dependencies between components by the number of “possible” dependencies between components.

Intuitively, when this ratio is low (e.g., only a limited percentage of all possible dependencies is defined) a change is more likely to remain local since there is only a limited number of dependencies over which it can propagate, which might indicate a higher effectiveness of encapsulation. This metric adheres to all criteria and is therefore included in our experiment.

The other two metrics defined by Briand et al. (1993) are the “Import Coupling” and the “Export Coupling” of a component. Because these metrics are defined at the level of components and cannot be calculated on the system level we do not include them in our experiment.

Lakos (1996) defines a metric called *Cumulative Component Dependency* (CCD), which is the sum of the number of components needed to test each component. On the system-level, this is equivalent to the sum of all outgoing dependencies of all components. Two variations are defined as well, the *Average Cumulative Component Dependency* (ACD) and the *Normalized Cumulative Component Dependency* (NCD). To arrive at the NCD, the CCD is divided by the number of components of the system. For ACD, the CCD is divided by the total number of modules in the system. Higher values of these metrics indicate that components are more dependent on each other, which increases the likelihood for changes to propagate. Thus a lower value for these metrics could indicate a more effective encapsulation. All three metrics adhere to all criteria and are therefore included in our experiment.

Mancoridis et al. (1998) propose a metric aimed at expressing the proper balance between coupling and cohesion in a modularization. Their metric is intended for steering a (search-based) clustering algorithm. Unfortunately, their metric takes into account an average coupling value. By using an average value, important information about outliers is lost during the calculation of the metric, which makes it hard to perform a root-cause analysis. Therefore, we do not include clustering metrics like these in our experiment.

Allen and Khoshgoftaar (1999) use information theory as a foundation for measuring coupling. Using a graph-representation of a system, the average information per node (i.e., the *entropy*) is determined and the total amount of information in the structure of the graph is calculating by a summation. The use of “excess entropy” (the average information in relationships) in the calculation of the metric makes it hard to perform root-cause analysis. The same reasoning applies to the metrics introduced by Anan et al. (2009). Because of this property we do not include these metrics in the current experiment.

Martin (2002) defines several metrics based on the concepts of *Afferent Coupling*, which is the number of different components depending on modules in a certain component, and *Efferent Coupling*, which is the number of components the modules of a certain component depend upon. Both of these metrics are defined at the level of components and are therefore not selected for our experiment. However, the metrics can be lifted to the system level. The result is that both metrics become equal to the number of dependencies between components, a metric which is considered in our experiment under the name Number of Binary Dependencies (NBD) as discussed at the end of this section.

Sant’Anna et al. (2007) defines a set of concern-driven metrics. Part of the input of these metrics is a mapping between the source code and functional concerns (e.g., "GUI" or "Persistence") implemented in the system, which is not available in our operationalization of criterion C3. Moreover, the metrics are either defined at the level of a concern or at the level of a component, thus we exclude these metrics from the current experiment.

Sarkar et al. (2007) define an extensive set of architecture-level metrics. To start, they define the *Module Size Uniformity Index* which measures the distribution of the overall size of the system over the components. This metric adheres to five out of the six criteria for metrics we use, but its primary goal is to quantify the analyzability of a system; not encapsulation. Nonetheless, we include the metric in our experiment as a control variable. The accompanying metric, *Module Size Boundness Index* relies on an (unspecified) upper limit for the size of components. Because this upper limit is not available the metric cannot be calculated. Additionally, the aggregation to system-level requires an additional (user-specified) limit. Determining the best values for these parameters is considered to be a separate study, therefore we exclude the metric from our current experiment.

Another metric which is defined by Sarkar et al. (2007) is the *Cyclic Dependency Index*, a metric which quantifies the number of cycles in the component dependency graph. Similar to the other dependency based metrics, a higher value of this metric potentially leads to a higher degree of propagation of changes and thus indicates little effective encapsulation. Because this metric adheres to all criteria it is included in the current experiment. Lastly, Sarkar et al's *Normalized Testability Dependency Metric* is equivalent to Briand's RCI, and included under that name in our experiment.

Other metrics defined by Sarkar et al. (2007) such as the *API Function Usage Index* or the *Non-API Function Closedness* rely on the definition of a formal API of the components of a system. Other metrics such as the *The Layer Organization Index* or the *Concept Domination Metric* rely on a mapping of components to layers, or on a mapping of functional concerns to source-code. As stated in the operationalization of criterion C3 this information is not available due to the manual effort needed to create and maintain this information. Therefore we exclude these metrics from our experiment. In addition, Sarkar et al. extend their list of architecture metrics towards object-oriented concepts in (Sarkar et al., 2008). Because these metrics rely on paradigm-dependent concepts we do not consider these metrics in our experiment.

Sangwan et al. (2008) define a metric called *Excessive Structural Complexity* which combines low level complexity with a quantification of higher level dependencies. Unfortunately, the normalization involved in the calculation of the metric prevents a straight-forward root-cause analysis. Because of this property we do not include this metric in the current experiment.

In Chapter 6 we have defined *Component Balance*, an architecture-level metric which combines the number of components and the relative size of the components. Because this metric is designed to quantify the analyzability of a software system rather than the encapsulation this metric is included in the current experiment as a control variable.

In Chapter 7 we have introduced the concept of a *Dependency Profile* aimed at quantifying the level of encapsulation within a system. This is done by categorizing all code of a system as either being internal to a component, meaning that it is not called from or depends on code outside its own component, or being part of *required interface* of a component (*outbound* code plus *transit* code, representing code which calls code from other components) or the *provided interface* of a component (*inbound* code plus *transit* code, representing code which is called from other components). These metrics adhere to all criteria (see Chapter 7), thus we take into account the percentage of code in the provided interface (PI), the percentage of code in the required interface (RI) and the percentage of internal code (IC).

Lastly, several standard architecture metrics are taken into account. As a start the number of components (NC) is considered to be a candidate to represent the size of the architecture. Similar to Component Balance this metric does not seem to target encapsulation directly but is added to the experiment as a control variable for the experiment.

Furthermore, the number of dependencies between the components is also considered. Systems with a higher number of dependencies are expected to have more propagation of changes. Note that for the number of dependencies the number of binary dependencies between components (i.e., 1 if there are dependencies, 0 if there are no dependencies) or the precise number of dependencies between components (i.e., each dependency between modules is counted separately) can be taken into account. The first metric is defined as the number of binary dependencies (NBD) and included in the experiment under that name. The latter metric, however, is not included in the current experiment because this absolute number is influenced by the volume of a component.

8.3.4 Selection result

Table 8.2 provides a summary of the metrics suite for encapsulation that we will investigate in our experiment. The first nine directly address encapsulation, and adhere to all criteria. The last three do not directly address encapsulation, but are included as control variables for our experiment. Note that the metrics which are excluded may still be considered to be suitable for specific situations, but are out of scope of the current experiment and thesis. However, we envision additional experiments in which these metrics are addressed specifically as part of future work.

8.4 Experiment Design

The central question of this chapter is which software architecture metrics can be used as an indicator for the effectiveness of encapsulation of an implemented software architecture. To answer this question, we need to determine whether the metrics listed

Name	**Abbr.**	**Src.**	**Description**	**Aim**	**Ctrl.**
Ratio of Cohesive Interactions	RCI	(Briand et al., 1993)	Division of known interactions by possible interactions	low	
Cumulative Component Dependency	CCD	(Lakos, 1996)	Sum of outgoing dependencies of components	low	
Average CCD	ACD	(Lakos, 1996)	CCD divided by number of modules	low	
Normalized CCD	NCD	(Lakos, 1996)	CCD divided by the number of components	low	
Cyclic Dependency Index	CDI	(Sarkar et al., 2007)	Normalized number of cycles in the component graph.	low	
Provided Interface	PI	(Bouwers et al., 2011c)	Percentage of code which is dependent upon from other components	low	
Required Interface	RI	(Bouwers et al., 2011c)	Percentage of code which depends on code from other components	low	
Internal code	IC	(Bouwers et al., 2011c)	Percentage of code which is internal to a component	high	
Number of Binary Dependencies	NBD		The number of binary dependencies within a dependency graph	low	
Component Balance	CB	(Bouwers et al., 2011a)	Combination of number of components and their relative sizes	high	x
Module Size Uniformity Index	MSUI	(Sarkar et al., 2007)	Normalized standard deviation of the size of the components	low	x
Number of components	NC		Counts the number of components in a dependency graph	low	x

Table 8.2: *Architecture-level metrics suitable for use in software architecture evaluations*

in Table 8.2 are indicative for the effectiveness of encapsulation within a system. This is done by performing an empirical study in which historical data is used to analyze the effectiveness of encapsulation within a system in the past. We then try to correlate this effectiveness with the values of the selected metrics.

Since this type of evaluation of system-level architectural metrics has not been done before (Koziolek, 2011) we first define how the effectiveness of encapsulation can be measured in Section 8.4.1. Next, we define how metrics based on a single snapshot of a system and a metric based on changes between snapshots can be compared in Section 8.4.2 and Section 8.4.3. The procedure for correlating the different metrics is discussed in Section 8.4.4 and augmented in Section 8.4.5. Lastly, Section 8.4.6 provides a summary of the steps in the experiment.

In the design of the experiment we use the term *snapshot-based* metric to refer to metrics which are calculated on a single snapshot of a system (e.g., the number of components on a specific point in time). All metrics listed in Table 8.2 belong to this category. A metric which is calculated based on changes between snapshots of a system, for example the number of files that changes between two snapshots of a system, is referred to as a *historic* metric.

8.4.1 Measuring Historic Encapsulation

The process of encapsulation revolves around localizing the design decisions which are likely to change (Booch, 1994) (a process which is also known as “information hiding” (Parnas, 1972)). In the context of software architectures, measuring whether the changes made to a system are mainly local or spread throughout the system can be determined by looking back at the change-sets of a system.

In an ideal situation, a software system consists of components which are highly independent, encapsulating the implementation details of the functionality they offer. In this situation, a change to a specific functionality only concerns modules within a single component, which makes it easier to analyze and test the change made. Naturally, it is not expected that all change-sets of a system concern only a single component. However, a system which has a high level of encapsulation (i.e., in which the design decisions which are likely to change are localized) is expected to have more localized changes compared to a system in which the level of encapsulation is low.

In this experiment, a change-set is defined as a set of modules (see Section 8.3.1) which are changed together in *a unit of work* (e.g., a task, a commit or a bug-fix). Using the concepts of Yu et al. (2010) as a basis, each change-set is categorized as either *local* (all changes occur within a single component) or *non-local* (multiple components are involved in the change).

A *change-set series* is a list of consecutive change-sets representing all changes made to a system over a period of time. Note that a series of change-sets does not necessarily contain change-sets which belong together, it can very well be that each change-set concerns a different bug-fix. Our key-measure of interest is the ratio of change-sets in a series that is local: the closer this ratio is to one, the better the system was encapsulated.

More formally, let $S = \langle M, C \rangle$ be a system, consisting of a set of modules M and a set of components C. Each module is assigned to a component and none of the components overlap. More formally, the set $C \subseteq \mathcal{P}(M)$ is a partition of M, i.e.,

- $\forall c_1, c_2 \in C : c_1 \neq c_2 \Rightarrow c_1 \cap c_2 = \emptyset$ (no overlap)
- $\bigcup_{c \in C} = M$ (complete coverage)

For each module $m \in M$ the containing component is obtained through a function

- $component : M \to C$.

A change-set $cs = \{m_1 \dots m_n\}$ is a set of modules which have been changed together. For a change-set series $CS_s = (cs_1, cs_2, \dots, cs_m)$ we can determine for each change-set whether it is local by counting the number of components touched in this change-set, i.e., a change-set is local if and only if:

- $isLocal(cs) \Leftrightarrow |\{c | m \in cs \wedge c = component(m)\}| = 1$.

Given this property, the *ratio of local change* can be calculated by a division of the number of local change-sets by the total number of change-sets in a series:

- $ratioOfLocalChange(CS_s) = \frac{|\{cs | cs \in CS \wedge isLocal(cs)\}|}{|CS|}$

In our experiment, we consider a change-set series with a high ratio of local change-sets to represent a high degree of effectiveness in the encapsulation of the system. Note that it is possible to split up a change-set series into multiple series to obtain insight into the effectiveness of encapsulation for a certain period of time. However, to obtain an accurate representation of the effectiveness of encapsulation it is important that the number of change-sets in a change-set series is large enough to calculate a meaningful ratio. Therefore, the use of longer change-set series, e.g., covering a longer period of time, is advised.

8.4.2 Snapshot-based versus Historic

To compare a snapshot-based metric, e.g., the number of components of a system, against a historic metric, e.g., the ratio of local change, two input parameters need to be defined: the exact moment of the snapshot for the snapshot-based metric and the change-set series for the historic metric. To increase the accuracy of the calculation of the historic metric, the change-set series should be as long as possible. On the other hand, the value of the snapshot-based metric needs to be representative for the chosen change-set series, e.g., it should be possible to link each change-set in the series to the value of the snapshot-based metric.

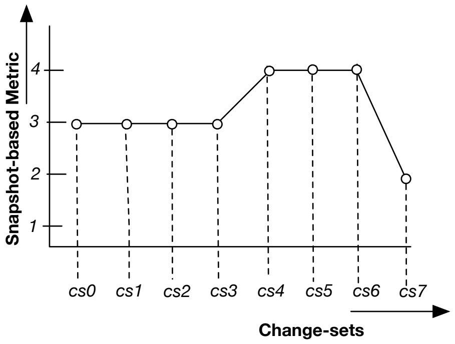


Figure 8.2: *The value of a snapshot-based metric over time determines the change-set series on which the historic metric should be calculated.*

In our experiment, this balance is obtained by calculating the historic metric using a change-sets series for which the snapshot-based metric is stable. To illustrate, consider the situation as shown in Figure 8.2 which shows the possible behavior of a snapshot-based metric given a series of change-sets. Instantiating this hypothetical graph with, for example, the number of components of a system we can see that this number is stable for some periods, but also changes over time. This means that we cannot use the complete change-set series $(cs_0,\ldots,c_7)$ to calculate a historic metric since there is no single snapshot-based metric value we can compare against. However, the value of a historic metric based on the two change-set series (cs_0,cs_1,cs_2,cs_3) and (cs_4,cs_5,cs_6) can be meaningfully compared against the values of the snapshot-based metric (respectively 3 and 4).

Note that this approach deviates from the commonly used design (see for example the experiments described by Yu et al. (2010) and Romano and Pinzger (2011)) in which a recent snapshot of the system is chosen and the historic metric is calculated based on the *entire* history of a system. The implicit assumption which is made in these experiments is that the value for the snapshot-based metric calculated on the specific snapshot is relevant for all changes throughout the history of the system. However, this assumption is not valid in all situations, or should at least be verified to ensure that the comparison between these two types of metrics is meaningful.

8.4.3 Metric Stability

One of the parameters that needs to be instantiated for this approach is the definition of when a metric has changed significantly. For some metrics this definition is straight-forward, e.g., any change in the number of components is normally considered to be significant from an architectural point of view. However, for metrics defined on a more continuous scale, such as for example RCI, the definition is less straight-forward.

The definition of when a metric changes has an impact on the conclusions that can be drawn from the data and the length of the change-set series for a metric. In general, the definition of which change in the value of a metric is significant is most-likely dependent on both the variability of the metric and the context in which the metric is used. For example, the number of components is not expected to change in a maintenance setting, while during the early stages of development it is expected that this number fluctuates heavily. In the implementation of the experiment a definition of “stable” related to the context of our goal is chosen.

8.4.4 Statistical Analysis

Given this approach, the aim of the experiment is to see whether the architecture metrics listed in Table 8.2 are correlated with the ratio of local change. To this end, we first define a null hypothesis for each of the twelve metrics that the desired value for the metric (for example a low number for the number of components or a high percentage of internal code) is not associated with a high (or low) ratio of local change.

To determine whether a null hypothesis can be rejected we perform a correlation test between the values of the snapshot-based metric and the ratio of local change. Because we cannot assume a normal distribution in any of the metrics, the specific correlation test used here is the Spearman rank correlation. Furthermore, because the hypotheses are directional a one-tailed test is performed. When the correlation test shows a moderate to strong correlation the null-hypothesis can be rejected, meaning that for a specific snapshot-based metric the values are correlated with the ratio of local change.

Using the thresholds defined by Hopkins (2000) we consider a significant correlation higher than 0.3 (or lower than -0.3) to indicate a moderate correlation, while a significant correlation score higher than 0.5 (or lower than -0.5) indicates a strong correlation. For a correlation to be significant the p-value of the test needs to be below 0.01, indicating that the chance that the found correlation is due to random chance is less than 1 percent.

In this set-up, multiple hypotheses are tested using the same dataset. In this case a Bonferroni correction (Hopkins, 2000) is appropriate to prevent the finding of a correlation by chance simply because one performs many different tests. The correction that needs to take place is the multiplication of the p-value of each individual test by the number of tests performed. If the resulting p-value is still below 0.01 the result of the test can be considered significant. Note that the use of the Bonferroni correction might lead to false negatives, i.e., not rejecting a null hypothesis even though there is a correlation. Our approach here is to be conservative by applying the correction. The impact of this choice is discussed in Section 8.8.3.

8.4.5 Preventing Project Bias

In this set-up data-points from several projects are combined into a single data-set to derive a correlation, instead of calculating the correlation on a per project basis.

This is primarily done because we are interested in a general trend across projects. Additionally, architecture-level metrics are expected to remain stable for long periods of time, resulting in just a few data-points per project, which makes it hard to derive statistically significant results.

However, it is possible that a single system contributes a proportionally large number of data-points to the sample used for correlation. If this is the case a significant correlation might be found just because the correlation occurs within a single system. To determine the impact of this issue a multiple regression analysis will be performed for all significant correlations.

The input of such an analysis is a linear model in which the dependent variable (i.e., the ratio of local change) is explained by a set of independent variables (i.e., the value of one of the snapshot-based metrics) plus one dummy variable per project. To determine which independent variables are significant a stepwise selection using a backward elimination model is applied (Hopkins, 2000). This process iterates over the input model and eliminates the least significant independent variables from the model until all independent variables are significant. If the resulting model contains the snapshot-based metric as the most significant factor (expressed by the R-squared value of the individual factor) we can conclude that the influence of the specific projects is negligible.

8.4.6 Summary

To summarize, the procedure for testing the correlation between each snapshot-based architecture metric and the historic ratio of localized change becomes:

- **Step 1:** Define when a metric is considered stable
- **Step 2:** Determine the change-set series for which the snapshot-based metric is stable on a per project/per metric basis
- **Step 3:** Calculate the historic metric for all selected change-set series
- **Step 4:** For each metric, calculate the correlation between the snapshot-based metric value and the historic metric using data from all projects
- **Step 5:** Verify the influence of individual projects on significant correlations

8.5 Experiment Implementation

8.5.1 Metric Stability

The context of this experiment is that of software analysts and software quality evaluators. In such a context it is useful to place a system within a bin according to its metrics in terms of different categories, i.e., 4 is a low number of components, 8 is a moderate number of components and 20 is a large number of components. A change in metric is interesting (and thus significant) as soon as the value of the metric shifts from one bin to another.

However, for most of the snapshot-based metrics there is no intuitive value which indicates when a systems should be placed in the "low", "moderate" or "large" category. Therefore, we take a pragmatic approach by defining a bin-size of 1. For example, if the number of components for a system on $(t_1, t_2, t_3, t_4, t_5)$ is $(4, 4, 5, 6, 6)$, the stable periods are considered to be $t_1 - t_2$ and $t_4 - t_5$.

Similarly, for percentage- and ratio metrics a change is considered significant as soon as the value is placed in a different bin, with a bin-size of 0.01. For example, when the values of the metric on snapshots (t_1, t_2, t_3) are (0.243, 0.246, 0.253), the snapshots t_1 and t_2 are considered to be equal, while snapshot t_3 is considered to be a significant change. The implications of choosing this bin-size are discussed in Section 8.8.2.

8.5.2 Stable Period Identification

To determine the time-periods for which a snapshot-based metric is stable, the value of the metric must calculated for different snapshots of the system. To obtain the most accurate result a snapshot should be taken after each change-set. However, given the large number of change-sets this approach requires an enormous amount of calculation effort. In order to compromise between precision and calculation effort a sampling approach is used.

Snapshots of the system are extracted on regular intervals, i.e., on every first day of the month, and all change-sets in between snapshots for which the snapshot-based metrics are stable are grouped together into a single change-set series. If the value of the snapshot-based metric changes significantly (as defined in Section 8.5.1) in between snapshots t_n and t_{n+1} all change-sets up until snapshot t_n are grouped into a single change-set series, while all change-sets in between t_n and t_{n+1} are discarded.

Note that for a highly unstable metric the effect may be that all data-points are discarded. This side-effect is not seen as a problem in this experiment because our aim is to identify metrics which can be used to steer development efforts, which requires the metric to be stable for a considerable period of time to enable the definition of corrective actions.

Also note that a change in the value of the snapshot-based metric indicates a change in the architecture of the system. Because our study only takes into account stable periods the study is focussed on determining the *effect* of these architectural changes, instead of the nature of the the architecture changes themselves. Even though we consider the actual architecture changes out of scope of the current thesis, we envision a thorough exploration of the unstable periods of a metric to be part of future work.

In this experiment we take one snapshot for each month that the system is active, i.e., changes are being made to the code. The snapshots are obtained from the source-code repository on the first day of each month. A more fine-grained interval (for example every week) might provide more accurate results, but since architecture metrics are not expected to change frequently a monthly interval is expected to be sufficient. The consequences of the decision for a sampling approach and the chosen sample-size are discussed in Section 8.8.2.

8.5.3 Subject systems

We used the following guidelines to determine the set of subject systems:

- *Considerable length of development*: at least a year's worth of data needs to be available in order to provide a representative number of change-sets.
- *Subversion repository*: this source-code repository system facilitates easy extraction of individual change-sets by assuming that each commit is a change-set.
- *Written in Java*: although the metrics are technology independent we have restricted ourselves to the Java technology because tool-support for calculating metrics for Java is widely available.

While choosing the systems we ensured that the set contains a mix of both library projects as well as complete applications. Table 8.3 lists the names of the systems used together with the overall start date and end date considered. The last two columns show the size of the subject system on respectively the start date and the end date to show that the systems have indeed changed over time.

8.5.4 Architectural Model Instantiation for Java

Following our earlier design from Chapter 6 we define the components of a system to be the top-level packages of a system (e.g., `foo.bar.baz`, `foo.bar.zap`, etc). Direct call relations between modules are taken as dependencies. Virtual calls (for example created by polymorphism or by interface implementations) are not considered to be a dependency. In other words, a call to an interface creates a dependency on that interface, but not on classes implementing that interface. All metrics are calculated on relevant source-code modules using the Software Analysis Toolkit of the Software Improvement Group (SIG)[1]. In this experiment, a module is considered to be relevant if it is production code which resides in the main source-tree of the system. Code written for testing or demo purposes is considered to be out of scope for this experiment (and therefore not included in the numbers of Table 8.3).

[1] `http://www.sig.eu`

		Period		**Size (KLOC)**	
Name	**Repository**	**Start**	**End**	**Start**	**End**
Ant	http://svn.apache.org/repos/asf/ant/core/trunk	2000-02	2011-05	3	97
Argouml	http://argouml.tigris.org/svn/argouml/trunk/src/argouml-app	2008-03	2011-07	113	108
Beehive	http://svn.apache.org/repos/asf/beehive/trunk/	2004-08	2008-10	45	86
Crawljax	http://crawljax.googlecode.com/svn/trunk/	2010-01	2011-07	6	7
Findbugs	http://findbugs.googlecode.com/svn/trunk/findbugs	2003-04	2011-07	7	97
Jasperreports	http://jasperforge.org/svn/repos/jasperreports/trunk/jasperreports	2004-01	2011-08	28	171
Jedit	https://jedit.svn.sourceforge.net/svnroot/jedit/jEdit/trunk	2001-10	2011-08	35	79
Jhotdraw	https://jhotdraw.svn.sourceforge.net/svnroot/jhotdraw/trunk/JHotDraw	2001-03	2005-05	8	20
Lucene	http://svn.apache.org/repos/asf/lucene/dev/trunk/lucene/	2001-10	2011-08	6	67
Struts2	http://svn.apache.org/repos/asf/struts/struts2/trunk	2006-06	2011-07	25	22

Table 8.3: *Subject systems used in the study*

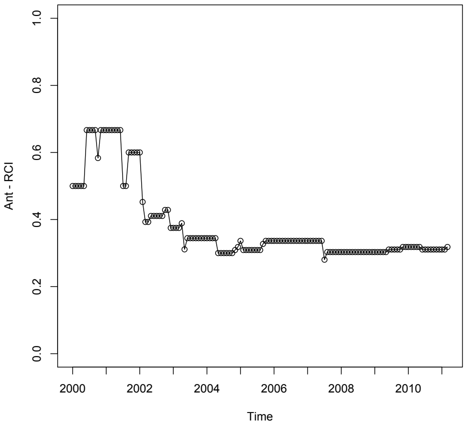


Figure 8.3: *The value of the RCI metric plotted over the life-time of the Ant system*

8.6 Experiment Results

8.6.1 Experiment Package

The raw data of the experiment is available in an on-line experiment package located at:

`http://www.sig.eu/en/QuantifyingEncapSA`.

This package contains:

D1: The descriptions of the top-level components and the scope used for each project

D2: The data-sets containing the change-sets with relevant modules used as an input to calculate the ratio of local change for a given period

D3: The data-sets listing the values of the snapshot-based metric for each month in which changes to the system have been made used to determine the stable periods

D4: The result of combining data-set D2 and D3 using a bin-size of 1/0.01

8.6.2 Stable Periods

The first step in the experiment is to determine the stable periods for each of the twelve snapshot-based metrics. To illustrate the need for determining these stable periods the value for the metric RCI for the system Ant is plotted Figure 8.3. This graph shows that there is considerable fluctuation in this metric during the development of the system, resulting in a total of 15 stable periods.

Descriptive statistics of the stable periods per metric illustrating four important characteristics of the data-set are shown in Table 8.4.

First of all, in the second column of Table 8.4 the number of stable periods per metric is shown. In all cases this number exceeds the number of projects (10), from which we can conclude that each metric changes over time and thus can be used to distinguish different states of a system.

Secondly, descriptive statistics of the number of months per stable period are shown in columns $3 - 5$ of Table 8.4. As discussed in Section 8.5.2 it is desirable for a snapshot-based metric to remain stable for a considerable period of time to enable the definition of corrective actions. We observe that the median number of months in a stable period variates between two and six months, with higher values up to three to six years. Even though on the low end there exists stable periods that last only a single month, we consider such a time-frame to be long enough to define corrective actions.

Thirdly, the sixth column of Table 8.4 shows the percentage of development time which is covered by the months in all stable periods. We observe that this percentage is at least 65% for all metrics, which means that more than half of the total development time of the systems is covered by the stable periods. From this we conclude that the metrics are stable enough to be used in the context of our experiment.

Lastly, columns $7 - 11$ of Table 8.4 show descriptive statistics for the length of the change-set series based on the stable periods. As discussed in Section 8.4.2 it is desirable to have longer change-set series to ensure an accurate representation of the ratio of local change. However, Table 8.4 shows that there are change-set series containing only a single change-set, which means that the ratio of local change will either be one or zero. When many change-sets series contain only a few change-sets the accuracy of the ratio of local change could be considered inadequate. However, as column 11 shows that for all metrics at least 91% of the change-sets series contain more than ten change-sets (up to over 5000 change-sets), we consider these change-sets series to be accurate enough and use all of the series in the current experiment.

8.6.3 Ratio of local change

For each of the snapshot-based metrics the ratio of local change is calculated based on the stable periods described in Table 8.4. Table 8.5 shows descriptive statistics of the result of this calculation. We observe that all metrics show considerable variation in the ratio of local change. The central tendency of the ratio of local change appears to be close to 0.85 for all metrics. This indicates that it is common to make more local than non-local changes during periods in which a snapshot-based metric is stable, which is inline with the expectations that design decisions that change often are indeed encapsulated.

In this chapter we do not study the ratio of local change within the *unstable* periods. We suspect that the ratio of local change during unstable periods tends to

		Months				change-sets series length				
Metric	**periods**	**Min**	**Med.**	**Max**	**covered**	**Min**	**Med.**	**Max**	**total**	> 10
RCI	94	1	4.0	38	80.9 %	3	113.0	968	17760	93.6 %
CCD	71	1	6.0	40	85.9 %	3	222.0	1178	19011	97.2 %
ACD	111	1	3.0	38	75.6 %	1	92.0	954	16564	91.9 %
NCD	74	1	4.5	40	83.6 %	3	192.5	1174	17922	95.9 %
CDI	65	1	6.0	50	88.3 %	1	224.0	2334	20526	95.4 %
PI	122	1	3.0	35	68.1 %	3	67.5	715	13811	95.9 %
RI	111	1	3.0	42	71.8 %	3	68.0	1337	15346	94.6 %
IC	119	1	2.0	41	71.2 %	2	50.0	1257	14759	91.6 %
NBD	108	1	3.0	38	75.8 %	3	88.5	846	15436	94.4 %
CB	82	1	3.0	77	80.6 %	3	76.5	5147	19345	91.5 %
MSUI	99	1	3.0	35	77.1 %	1	91.0	1176	18028	93.9 %
NC	59	1	6.0	53	90.8 %	7	262.0	1805	21428	96.6 %

Table 8.4: *Descriptive statistics for the stable periods per snapshot-based metric*

be lower than during stable periods, but an in-depth analysis of this hypothesis is considered to be out of scope for the current thesis.

8.6.4 Correlation values

Using the values of the snapshot-based metrics and the ratio of local change we calculate the Spearman rank correlation between the two samples. Table 8.6 shows the results of the tests together with both the corrected as well as the original p-values.

As can be seen from the results, many of the correlation tests do not result in a significant correlation. This result is expected for the control variables MSUI, CB and NC, but for the other metrics a lack of significance is unexpected. One reason for this result could be that the number of data-points in the sample for a metric is not large enough to detect correlation. However, looking at the size of the samples as displayed in the second column of Table 8.4 this is not likely. Moreover, using a power t-test to determine the required sample size needed to find correlation shows that all samples contain enough data-points (Hopkins, 2000).

For PI, RI and the IC metric the result of the correlation tests is significant even after applying the correction. For these three metrics we performed a multivariate regression analysis to determine whether any of the projects have a significant influence on the found correlation (see Section 8.4.5 for details about this approach). The resulting models for respectively PI, RI and IC are listed in table Table 8.7, 8.8 and 8.9. In all three cases some of the projects have a significant influence (the other projects are eliminated from the model because their influence is non-significant), but the value of the snapshot-based metric is the most significant factor (see the last column) in each model. In other words, the snapshot-based metric explains most of the variation in the ratio of local change.

	Ratio of local change		
Metric	**Min**	**Median**	**Max**
RCI	0.00	0.84	1
CCD	0.37	0.84	1
ACD	0.00	0.85	1
NCD	0.37	0.84	1
CDI	0.00	0.84	1
PI	0.24	0.86	1
RI	0.25	0.86	1
IC	0.16	0.86	1
NBD	0.00	0.84	1
CB	0.35	0.86	1
MSUI	0.36	0.84	1
NC	0.18	0.83	1

Table 8.5: *Descriptive statistics of the ratio of local change calculated on the stable periods described in Table 8.4*

Metric	**Correlation**	**Corrected p-value**	**p-value**
RCI	0.16	11.3	0.94
CCD	-0.27	0.13	0.01
ACD	-0.26	0.04	< 0.01
NCD	-0.19	0.59	0.05
CDI	0.32	11.94	1.00
PI	**-0.30**	< 0.01	< 0.01
RI	**-0.31**	< 0.01	< 0.01
IC	**0.47**	< 0.01	< 0.01
NBD	-0.22	0.14	0.01
CB	0.29	0.05	< 0.01
MSUI	-0.08	2.42	0.20
NC	-0.26	0.27	0.02

Table 8.6: *Correlation values between each snapshot-based metric and the ratio of local change*

Model	**Coef.**	**Std. Error**	t	p**-value**	R^2
(Constant)	1.032	0.033	31.114	< 0.01	–
PI	-0.616	0.079	-7.811	< 0.01	0.275
Beehive	-0.321	0.073	-4.419	< 0.01	0.072
Findbugs	0.087	0.030	2.843	< 0.01	0.032
Lucene	-0.065	0.032	-2.011	< 0.05	0.022
Model Summary: $R2 = 0.4006; p \leq 0.01$					

Table 8.7: *Linear model for predicting the ratio of local change using the percentage of code in the Provided Interface and specific projects*

Model	**Coef.**	**Std. Error**	t	p**-value**	R^2
(Constant)	0.987	0.036	27.528	< 0.01	–
RI	-0.460	0.061	-7.517	< 0.01	0.167
Ant	0.253	0.038	6.720	< 0.01	0.094
ArgoUML	0.245	0.043	5.650	< 0.01	0.067
Findbugs	0.180	0.031	5.818	< 0.01	0.064
Lucene	0.145	0.026	5.509	< 0.01	0.080
Model Summary: $R2 = 0.4733; p \leq 0.01$					

Table 8.8: *Linear model for predicting the ratio of local change using the percentage of code in the Required Interface and specific projects*

Model	**Coef.**	**Std. Error**	t	p**-value**	R^2
(Constant)	0.564	0.030	19.200	< 0.01	–
IC	0.641	0.076	8.427	< 0.01	0.247
Ant	0.228	0.039	5.873	< 0.01	0.074
ArgoUML	0.139	0.046	2.984	< 0.01	0.013
Beehive	-0.190	0.087	-2.191	< 0.05	0.023
Findbugs	0.199	0.037	5.334	< 0.01	0.057
Lucene	0.107	0.027	3.843	< 0.01	0.051
Model Summary: $R2 = 0.4672; p \leq 0.01$					

Table 8.9: *Linear model for predicting the ratio of local change using the percentage of Internal Code and specific projects*

8.7 Discussion

8.7.1 Key Findings

The results of the experiment shows that there is not enough evidence to reject the null hypothesis for the metrics RCI, CCD, ACD, NCD, NBD, MSUI, CDI, CB, and NC. For PI, RI and IC the found correlation is moderate, thus the null hypothesis may be rejected. In other words, the results of the experiment shows that *the percentage of code in the provided interface*, *the percentage of code in the required interface* and *the percentage of internal code* are correlated with *the historic ratio of local change*.

In our opinion, the reasoning behind this correlation lies in the fact that these metrics accurately quantify the requires interface, the provided interface, and the non-interface code of the components of a system. The larger the interfaces of a component, the more likely it is that changes in one place will propagate to other components. From this point of view, these metrics are measuring the extent, e.g., the "width", of the connection between components instead of only the intensity of these connections.

Note that the first two results provide evidence for the two hypothesis as defined in Section 7.6, while the last result was not originally expected. However, despite the relationship between the metrics, e.g., a larger requires interface automatically leads to a lower percentage of non-interface code, we observe that the quantification of all non-interface code provides a stronger correlation than a quantification of either the required or the provided interfaces within a system. Moreover, the percentage of internal code is more closely related to the notion of encapsulation as defined in Section 8.4.1. Based on these observations the answer to our research question becomes:

> *The percentage of internal code can serve as an indicator for the effectiveness of encapsulation of an implemented software architecture.*

8.7.2 Beyond "boxes and arrows"

The results show that for nine metrics there is insufficient evidence to conclude that these metrics have indicative power for the level of encapsulation of a software architecture. For the three control variables (MSUI, CB and NC), this result can be attributed to a difference in goal. These metrics are primarily designed to quantify the analyzability of a system instead of the encapsulation.

The other metrics for which no correlations were found (i.e., RCI, CCD, ACD, NCD, CDI and NBD) are all based on a graph view (boxes and arrows) of the software architecture. Possibly, the inability of these metrics to measure encapsulation derives from the over-simplification inherent in such a view. More specifically, even though these metrics capture the intensity of the dependencies between components, we suspect that they are not able to properly quantify the extent of the dependency between components.

Since more sophisticated views apparently allow for the definition of more powerful metrics, we recommend investigating metrics based on even more sophisticated views. For example, one could augment the view with information on the entry points of the system (e.g., the external API for a library or user interface actions for an application) or on the participation of the various components in the implementation of various concerns. Construction of such views currently requires manual input. To make construction of such views feasible in the context of (repeated) evaluations, techniques must be found to lift such limitations. The first steps in this area have already been taken (Olszak et al., 2012).

8.7.3 Generalization

The current implementation of the experiment limits the generalizability of the results to open-source systems written in Java. In Chapter 7 we already investigated the behavior of the different categories of the dependency profiles on different technologies and found little variance between technologies, but replication of our experiment using systems with other characteristics (i.e., non object-oriented systems, industry systems) is needed to determine the exact situations in which the metrics are usable. Because the design of the experiment as described in Section 8.4 does not impose any limitations on the characteristics of the systems this can be done with relatively little effort.

Furthermore, the fact that we only examined the stable periods of these systems means that the indicative power of the metrics cannot be ensured while a system is undergoing large refactorings on the level of the architecture. We do not consider this a problem, since the snapshot-based metrics aim to quantify characteristics of the architecture of a system and are therefore expected to remain stable for longer periods. This is supported by the data in Table 8.4 which shows that the metrics are stable for an average period of at least two months, and are stable for more than 60% of the time a system is under development.

8.7.4 Implications for Architecture Evaluations

The implication of these results for late architecture evaluations is that the percentage of internal code can be used to reason about the level of encapsulation within a system. We envision that a low percentage of internal code could be a reason to steer the refactoring of a code base to internalize modules within components.

From an organizational perspective, this metric can be used to provide a first indication of the level of the encapsulation of many systems across an application portfolio. By combining the value of this metric with other key-properties of the systems (e.g., size, business criticality or expected change rate) the allocation of resources can be conducted on a more informed basis.

Based on the findings in this report, SIG (a consultancy firm specialized in the analysis of the quality of software systems) has decided to include, amongst others, the percentage of internal code in its suite of metrics used to conduct (repeated) architectural evaluations. The results of applying these metrics in over 500 industrial projects are available in Chapter 9.

8.7.5 Metric Stability

As can be seen in Table 8.4, the metrics measured on the level of the architecture of a system have the tendency to be stable for a period between two and six months. The implication of this finding is that the assumption that the value of a snapshot-based metric is representative for all changes that occurred during the entire history of a system is not correct. If this is the case on the system-level, this assumption must also be verified when these types of experiments are performed on the level of modules or units. An alternative solution is to explicitly encode these assumptions into the design of the experiment, as we have done in Section 8.4.

8.8 Threats to Validity

Following the guidelines of Wohlin et al. (2000) the threats to the validity of the results of the experiment are categorized in four categories addressing construct, internal, external and conclusion validity. Because the generalization of the results (external validity) has already been addressed in Section 8.7.3, this category of threats is not discussed in detail in this section.

8.8.1 Construct validity

The basis for our experiment is the assumption that the ratio of local change accurately models the concept of encapsulation. This relationship between encapsulation and the localization of change has been made explicit by, amongst others, Booch (1994). In addition, "encapsulate what changes" is a well known and widely recognized design principle (Gamma et al., 1995). Additionally, the control variables MSUI, CB and NC do not show any significant moderate to strong correlation, which limits the threat that the ratio of local change measures a different external quality attribute.

A second question regarding construct validity is whether the top-level packages of a software system can be used as the architectural components of a software system. We have performed many such componentizations for industry systems and validated them with the development teams. In many of these cases the top-level packages are indeed considered to be the top-level components. Thus, even though we did not perform an explicit validation of the component-structure with the developers of the systems in our experiment, the naming of the top-level packages seems to indicate a valid component structure.

A last question is whether the assumption that a commit into the Subversion source-code repository of the systems is a coherent unit of work is valid. This assumption might not hold for developers which have a particular style of committing code, for example always committing several fixes at once or committing changes made to each component in isolation. Although this effect might exist this threat is countered by taking into account several projects, and thus different developers with a different style of committing.

Additionally, it could be the case that a commit consists solely of automated refactorings such as renaming a class, or changing the license statement in comments only. We did not explicitly deal with these cases, but argue that the first type is not problematic since an automated refactoring which impacts multiple components is a legitimate non-local change-set. The second type might be problematic, but we consider the impact of this case to be minor due to the large number of change-sets used both per stable period and in total.

8.8.2 Internal validity

As discussed before, there might be confounding factors which explain the correlation between the snapshot based metric values and the ratio of local change. One of these confounding factors, the influence of specific projects on the significant correlations, has already been addressed in the design and results of the study.

A second confounding factor is the choice for taking monthly snapshots to determine the stable periods of the snapshot-based metrics. Taking longer or shorter periods could result in a different number of stable time-periods, which could influence the variance of the ratio of local change. However, as can be determined from the data in Table 8.4, the median number of months which are taken into account as a stable period is between two and six months and over 60% of the development time is covered by these periods. This shows that using a one month period between snapshots already covers a considerable portion of the development of the system, thus using a shorter period of time between snapshots is not immediately warranted.

A related issue is that the value of a snapshot-based metric could fluctuate significantly between two snapshots of the system, but is still considered to be equal because the value has not changed significantly on the first of the month. However, given the average length of the stable-periods this situation does not occur often enough to influence the results significantly.

A third factor is the pragmatic choice to consider a percentage stable as long as the value stays within the same bin with a bin-size of 0.01. As discussed before, taking a more narrow or a broader bin-size can lead to more (or less) variance in the snapshot based metric which in turn leads to more (or less) co-variance with the ratio of local change. Again, the median length of the periods, the number of change-sets per period and the variation in the metrics as shown in Table 8.4 do not indicate that the bin-size is too small or too large for any of the metrics.

Moreover, we find that determining the optimal threshold for each snapshot based metric is a new research topic in its own right. We hypothesize that the bin-size could be defined on the stability characteristics of the metrics determined by, for example, techniques taken from time-series analysis. Note that in our situation the pragmatic choice of bin-size can only cause false negatives, e.g., using a different bin-size might lead to finding significant correlations where there is none with the current bin-size. In contrast, changing the bin-size will not invalidate the correlations found with the current bin-size.

8.8.3 Conclusion validity

The final question is whether the conclusions drawn from the data are sound. On the one hand, the metrics for which we do reject the null-hypothesis might not be valid indicators. This would be the case when there is no rationale for the correlation between the value of the snapshot-based metric and the ratio of local change. However, as discussed in Section 8.7.1 there is a logical explanation for this correlation, thus the conclusions drawn from the data are valid.

On the other hand, the metrics for which we do not reject the null-hypothesis might actually be valid indicators. As discussed before, the application of a Bonferroni correction could cause false negatives. Inspecting the non-corrected p-values shown in Table 8.6 shows that without correction ACD and CB also provide significant correlations with $p < 0.01$. However, in both cases the correlation values is below 0.3 and thus considered to be low, which means that not rejecting the null hypothesis remains correct.

8.9 Related Work

The change-history of a software system has previously been used to, amongst others, validate designs (Zimmermann et al., 2003), predict source-code changes (Ying et al., 2004) and for predicting faults (Graves et al., 2000). The majority of this work is focussed on predicting which artifacts are going to change together, while our focus is on correlating snapshot-based metrics with historic metrics. Apart from this difference in goal, the artifacts which are considered are on a different level (i.e., file versus components) or of a different nature (i.e., code versus faults).

With respect to the topic of validating snapshot-based metrics against change history of a system there is again a large body of work. As mentioned before, many class level metrics have been validated extensively, see Lu et al. (2012) for an overview.

On the component level this type of validation has been done by Yu et al. (2010). In this experiment, the relationship between the external co-evolution frequency (e.g., non-local change) and several size and coupling-related metrics is investigated using the complete history of nine open-source projects.

However, we are not aware of any study which validates system-level architecture metrics against the change-history of a system. Because of this we considered the validation of system-level architecture metrics for measuring encapsulation as an unresolved problem.

8.10 Conclusion

The goal of this chapter is to determine which existing system-level architecture metrics provide an indication of the level of encapsulation of an implemented software architecture in the context of late architecture evaluations. The contributions of this chapter are:

- A selection of twelve existing system-level architecture metrics which are suitable candidates to be used in an late architecture evaluation context.
- An experiment design in which the value of these twelve metrics are correlated with the ratio of local change during periods for which the metrics are representative.
- A stability analysis on twelve metrics which shows that the variability of a metric needs to be taken into account when comparing snapshot-based metrics against metrics based on multiple snapshots of a system.
- Strong evidence that the percentage of internal code provides an indication of the effectiveness of encapsulation of an implemented architecture.

The key implications of these results are two-fold. First, the percentage of internal code is suitable to be used in the evaluation of an implemented software architecture. For technical stakeholders the correlation between these percentages and the ratio of local change is expected to be sufficient to justify using the metrics within a decision making process. However, based on our experience we expect non-technical stakeholders to require a stronger relation between these values and costs or operational risks. Determining this relationship is considered to be out of scope for this thesis.

Secondly, the results show that the twelve architecture metrics tend to be stable for a period of two to six months. This property needs to be taken into account in any experiment in which these specific snapshot-based metrics are correlated against metrics based on multiple snapshots of a system. More generally, the assumption that a snapshot-based metric is representative for the period of time on which an historic metric is calculated must be verified for any experiment in which these two types of metrics are correlated.

There are two main areas in which future work is needed. First of all, we plan on verifying the usefulness of the percentage of internal code by using this metrics in a late architecture evaluation setting. This is done by incorporating this metrics in the suite of metrics which is used by SIG to conduct (repeated) architectural evaluations. The design and results of this empirical study are available in Chapter 9.

Secondly, we envision a study aimed towards determining the best way to define the stability of software metrics. Such a study would not only improve the experiment design as proposed in this chapter, it would also help in interpreting metrics currently used in the monitoring of software systems. Because of the effort involved in properly investigating this subject we consider this topic to be out of scope for this thesis.

8.11 Architecture Metrics Overview

Table 8.10 provides an overview of the metrics considered in the evaluation of the metrics as described in Section 8.3. The column headers C1 to C6 are defined in Table 8.1 and repeated below:

C1: has the potential to measure the level of encapsulation within a system

C2: is defined at (or can be lifted to) the system level

C3: is easy to compute and implement

C4: is as independent of technology as possible

C5: allows for root-cause analysis

C6: is not influenced by the volume of the system under evaluation

Source	**Name**	**Abbr.**	**C1**	**C2**	**C3**	**C4**	**C5**	**C6**
Briand et al. (1993)	Ratio of Cohesive Interactions	RCI	Y	Y	Y	Y	Y	Y
Briand et al. (1993)	Import Coupling	ImC	-	N	-	-	-	-
Briand et al. (1993)	Export Coupling	ExC	-	N	-	-	-	-
Lakos (1996)	Cumulative Component Dependency	CCD	Y	Y	Y	Y	Y	Y
Lakos (1996)	Average CCD	ACD	Y	Y	Y	Y	Y	Y
Lakos (1996)	Normalized CCD	NCD	Y	Y	Y	Y	Y	Y
Mancoridis et al. (1998)	Modularization Quality	MQ	Y	Y	Y	Y	N	-
Allen et al. (1999)	Intramodule Coupling	INMC	Y	Y	Y	Y	N	-
Allen et al. (1999)	Cohesion of a Modular System	COMS	Y	Y	Y	Y	N	-
Anan et al. (2009)	Information Entropy of an Architectural Slicing	IEAS	-	N	N	-	-	-
Anan et al. (2009)	Coupling between Architecture Slicings	ASC	-	N	N	-	-	-
Anan et al. (2009)	System Coupling	SC	Y	Y	Y	Y	N	-

Continued on next page

Continued from previous page

Source	**Name**	**Abbr.**	**C1**	**C2**	**C3**	**C4**	**C5**	**C6**
Anan et al. (2009)	Architecture Slicing Cohesion	ASC	-	N	N	-	-	-
Martin (2002)	Afferent Coupling	Ca	-	N	-	-	-	-
Martin (2002)	Efferent Coupling	Ce	-	N	-	-	-	-
Martin (2002)	Instability	I	-	N	-	-	-	-
Martin (2002)	Abstractness	A	-	N	-	-	-	-
Martin (2002)	Distance from the Main Sequence	D	-	N	-	-	-	-
Sant'Anna et al. (2007)	Concern Diffusion over Architectural Components	CDAC	-	N	N	-	-	-
Sant'Anna et al. (2007)	Concern Diffusion over Architectural Interfaces	CDAI	-	N	N	-	-	-
Sant'Anna et al. (2007)	Concern Diffusion over Architectural Operations	CDAO	-	N	N	-	-	-
Sant'Anna et al. (2007)	Component-level Interfacing Between Concerns	CIBS	-	N	N	-	-	-
Sant'Anna et al. (2007)	Interface-level Interfacing Between Concerns	IIBC	-	N	N	-	-	-
Sant'Anna et al. (2007)	Operation-level Overlapping Between Concerns	OOBC	-	N	N	-	-	-
Sant'Anna et al. (2007)	Lack of Concern-based Cohesion LLC	IIBC	-	N	N	-	-	-
Sarkar et al. (2007)	Module Interaction Index	MII	-	-	N	-	-	-
Sarkar et al. (2007)	Non-API Function Closedness	NAFC	-	-	N	-	-	-
Sarkar et al. (2007)	API Function Usage Index	APIU	-	-	N	-	-	-
Sarkar et al. (2007)	Implicit Dependency Index	IDI	-	-	N	-	-	-
Sarkar et al. (2007)	Module Size Uniformity Index	MSUI	N	Y	Y	Y	Y	Y
Sarkar et al. (2007)	Module Size Boundedness Index	MSBI	N	Y	N	-	-	-
Sarkar et al. (2007)	Cyclic Dependency Index	CDI	Y	Y	Y	Y	Y	Y
Sarkar et al. (2007)	The Layer Organization Index	LOI	-	-	N	-	-	-
Sarkar et al. (2007)	Module Interaction Stability Index	MISI	-	N	-	-	-	-
Sarkar et al. (2007)	Normalized Testability Dependency Metric	NTDM	Y	Y	Y	Y	Y	Y
Sarkar et al. (2007)	Concept Domination Metric	CDM	Y	Y	N	-	-	-
Sarkar et al. (2007)	Concept Coherency Metric	CCM	Y	Y	N	-	-	-
Sarkar et al. (2008)	Base-Class Fragility Index	BCFI	Y	Y	Y	N	-	-
Sarkar et al. (2008)	Inheritance Based Coupling	IBC	Y	Y	Y	N	-	-
Sarkar et al. (2008)	Not-Programming-to-Interfaces Index	NPII	Y	Y	Y	N	-	-
Sarkar et al. (2008)	Association-induced Coupling	AIC	Y	Y	Y	N	-	-
Sarkar et al. (2008)	State Access Violation Index	SAVI	Y	Y	Y	N	-	-
Sarkar et al. (2008)	Plugin Pollution Index	PPI	Y	Y	Y	N	-	-

Continued on next page

Continued from previous page

Source	Name	Abbr.	C1	C2	C3	C4	C5	C6
Sarkar et al. (2008)	API Usage Index	APIU	Y	Y	Y	N	-	-
Sarkar et al. (2008)	Common Reuse of Modules	CReuM	Y	Y	Y	N	-	-
Sangwan et al. (2008)	Excessive Structural Complexity	XS	Y	Y	Y	Y	N	-
Bouwers et al. (2011a)	Component Balance	CB	N	Y	Y	Y	Y	Y
Bouwers et al. (2011c)	Internal Code	IC	Y	Y	Y	Y	Y	Y
Bouwers et al. (2011c)	Provided Interface	PI	Y	Y	Y	Y	Y	Y
Bouwers et al. (2011c)	Required Interface	RI	Y	Y	Y	Y	Y	Y
	Number of components	NC	Y	Y	Y	Y	Y	Y
	Number of Binary Dependencies	NBD	Y	Y	Y	Y	Y	Y
	Number of Absolute Dependencies	NAD	Y	Y	Y	Y	Y	N

Table 8.10: *The overview of a comparison of architecture metrics against the criteria as listed in Section 8.3 grouped per source*

CHAPTER 9

Evaluating Usefulness of Software Metrics - an Industrial Experience Report [*]

Abstract

A wide range of software metrics targeting various abstraction levels and quality attributes have been proposed by the research community. For many of these metrics the evaluation consists of verifying the mathematical properties of the metric, investigating the behavior of the metric for a number of open-source systems or comparing the value of the metric against other metrics quantifying related quality attributes.

Unfortunately, a structural analysis of the usefulness of metrics in a real-world evaluation setting is often missing. Such an evaluation is important to understand the situations in which a metric can be applied, to identify areas of possible improvements, to explore general problems detected by the metrics and to define generally applicable solution strategies.

In this chapter we execute such an analysis for two architecture level metrics, Component Balance and Dependency Profiles, by analyzing the challenges involved in applying these metrics in an industrial setting. In addition, we explore the usefulness of the metrics by conducting semi-structured interviews with experienced assessors. We document the lessons learned both for the application of these specific metrics, as well as for the method of evaluating metrics in practice.

[*] Originally published in the proceedings of the 35th International Conference on Software Engineering (ICSE 2013), Software Engineering in Practice (SEIP) track (Bouwers et al., 2013).

9.1 Introduction

Software metrics continue to be of interest for researchers and practitioners. Metrics such as volume (Albrecht and Gaffney, 1983), McCabe Complexity (McCabe, 1976), the C&K metric suite (Chidamber and Kemerer, 1994) and a wide range of architecture metrics (see Koziolek (2011) for an overview) are well-known and used in practice. Moreover, new software metrics continue to be defined by the research community.

The evaluation of a new metric typically consists of correlating the (change in) value of the metric with other quality indicators such as likelihood of change (Lu et al., 2012) or its ability to predict the presence of bugs (Nagappan and Ball, 2005). In other cases, the evaluation consists of an analysis of the values of a metric for a set of open-source systems, either on one single snapshot or over a period of time (Sarkar et al., 2007 & 2008). More theoretical approaches of metric evaluation inspect mathematical properties of metrics (see for example Briand et al. (1999b & a) and Fenton and Pfleeger (1998)) or focus on metrological properties of metrics (see for example Abran (2010)).

The focus of these types of evaluation is to determine whether the designed metric is related to the quality property it has been designed to quantify, a property known as “construct validity” (Kaner and Bond, 2004). Although this is an important part of the evaluation of a metric, these types of evaluations cannot be used to determine whether a metric is *useful*. For a metric to be considered useful its value should correspond to the intuition of an measurer (Fenton and Pfleeger, 1998) and should be actively used in a decision-making process (Gopal et al., 2005).

In this chapter we evaluate the usefulness of two architecture level metrics, Component Balance (see Chapter 6) and Dependency Profiles (see Chapter 7), which are designed to quantify the analyzability and encapsulation within a software system. Evidence of the construct validity of these metrics has been previously gathered in small-scale experiments presented in Chapter 6 and Chapter 8. The large-scale study presented here aims to gain an understanding of the usefulness of these two metrics in practice.

The context of this research is the Software Improvement Group (SIG), a consultancy firm specialized in providing strategic advice to IT management based on technical findings. As a first step both metrics are embedded in the measurement model used to monitor and assess the technical quality of a large set (500+) of systems developed by (or for) clients of SIG. The metrics are interpreted by consultants working at SIG, who fulfill the role of external quality assessors.

Data about the usefulness of the metrics is collected using two different methods. First, data about the challenges involved in actually applying the metrics is collected by observing the quality assessors and documented in the form of memos. Secondly, semi-structured interviews are conducted with the quality assessors who interpreted the metrics when assessing their customers’ software systems.

Our analysis of the collected data leads to two types of findings. First, we identify in which situations and under which conditions the metrics are useful. Second, we discover how to improve the metrics themselves and ways to apply them better.

In addition to reporting on the evaluation of these specific metrics in this particular context, we reflect upon a general method for evaluating software metrics in a practical setting. The challenges involved in designing and executing such a study are outlined and the generalizability of the results is discussed. We conclude that despite the inherent limitations of this type of studies, the execution of such a study is crucial for the proper evaluation of any software metric.

9.2 Evaluation Goal

The goal of this study is to gain an understanding of the usefulness of two software metrics. Before we can reason about the usefulness of a metric the term “usefulness” needs to be characterized. Different characteristics of usefulness are available, e.g. a metric is considered useful if the metric:

- corresponds to the intuition of the measurer (Fenton and Pfleeger, 1998)
- is actively used in a decision making process (Gopal et al., 2005)

In the first definition, a crucial role is played by the person using the metric. Apart from the experience of the particular person, the role in which he/she uses the metric, e.g., a developer, a quality assessor inside a company or an external quality assessor, has a significant impact on the outcome of the evaluation. In the second definition, the context in which the metric is used, e.g., assessing the quality of a system, analyzing the properties of an architecture or assessing the performance of developers, has a large influence on the outcome of the evaluation.

The subjects of this evaluation are the Component Balance (see Chapter 6) and Dependency Profile (see Chapter 7) metrics. In previous chapters have evaluated the properties, correlations and statistical behavior of these metrics, but not their usefulness.

Both of these metrics have been designed to quantify specific properties of the implemented architecture of a software system, i.e., analyzability and encapsulation. Such a quality assessment can either be done by an internal assessor (e.g., working inside a single company) or an external assessor (for example employed by an external consultancy firm). In this research, we choose the viewpoint of external assessors.

Thus we define our evaluation goal in the template from the GQM approach of Basili et al. (1994) as follows:

> *The objective of the study is to understand the usefulness of the “Component Balance” and “Dependency Profiles” metrics, from the point of view of external quality assessors, in the context of external assessments of implemented architectures.*

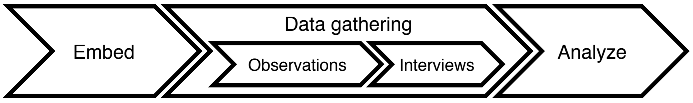


Figure 9.1: *A four-step process for evaluating software metrics in practice*

9.3 Evaluation Method

To answer our research question we use the four step methodology as outlined in Figure 9.1. To start, the metrics are included in the standard operating procedure of the external assessors. Details about this embedding and the context of the external assessors are given in Section 9.4.

In line with recommendations about collecting data in qualitative research (Adolph et al., 2011), data about the challenges involved in applying the metrics are gathered using two methods. First, an observer records real-word experiences of using the metrics in the form of memos. Secondly, interviews with the assessors are conducted to determine the perceived usefulness of the metric as seen by the assessors. Details about this data-gathering process are given in Section 9.5.

Lastly, the data extracted by both methods is analyzed and condensed separately, the results of which are given in Section 9.6 and Section 9.7. Based on this data, the most common observations are discussed and possible solution areas are explored in Section 9.8.

Apart from evaluating the metrics we also reflect upon the benefits and limitations of this evaluation process in Section 9.9, discuss related work in Section 9.10, after which the chapter concludes in Section 9.11.

9.4 Evaluation Setting

The evaluation took place within the Software Improvement Group (SIG), a consultancy firm which "... translates detailed technical findings about software-intensive systems into actionable information for top management".[1]

The length of the investigation was six months, from the start of February 2012 until end of August 2012. At the start of this period, two system properties based on the Component Balance and the Dependency Profiles metrics were added to the company's software measurement model. Details about this measurement model and the embedding of the metrics are given in Section 9.4.1 and Section 9.4.2.

The measurement model is applied by consultants employed by SIG on a wide range of customer systems throughout various services. Details about the consultants interpreting the metrics are discussed in Section 9.4.3, while the context of this

[1] http://www.sig.eu

research is described in two parts focussing on the services in which the metrics are used (Section 9.4.4) and the type of systems assessed in the services (Section 9.4.5).

9.4.1 Software Measurement Model

A measurement model based on the maintainability aspect of the ISO/IEC 9126 (International Organization for Standardization, 2001) standard for software quality was used throughout the services of SIG (Heitlager et al., 2007). This model operationalizes the standard by decomposing its sub-characteristics into a set of six system properties, which are quantified by code-level metrics such as the duplication percentage and length of units.

These code-level metrics are in turn used to derive a rating for each system property using a benchmarking-based methodology (Alves et al., 2011). More details about the exact construction of the model and its application can be found in Baggen et al. (2010).

Since the introduction of this model, the ISO/IEC 9126 standard has been replaced by the new ISO/IEC 25010 (International Organization for Standardization, 2011) standard for software quality. One of the changes of the new standard is the introduction of the sub-characteristic of "Modularization", which is defined as:

> [The] degree to which a system or computer program is composed of discrete components such that a change to one component has minimal impact on other components.

Taking into account the pitfalls introduced in Chapter 5, the measurement model was extended with two system properties to capture this new sub-characteristic: *Component Balance* and *Component Independence*. Apart from upgrading to the latest quality standard, the introduction of these system properties was expected to stimulate discussions about the architecture of systems and to incorporate a common view-point in the assessment of implemented architectures.

9.4.2 Component System Properties

The metric used to quantify the system property of Component Balance is the metric with the same name as was introduced in Chapter 6. The metric used to quantify the Component Independence system property is the combination of two categories of the Dependency Profiles metric introduced in Chapter 7. Both metrics were chosen based on the results of our earlier experiments, which showed that these metrics outperform other metrics when quantifying the quality characteristics of analyzability (see Chapter 6) and encapsulation (see Chapter 8).

Component Balance The Component Balance metric consists of two factors, System Breakdown (SB) and Component Size Uniformity (CSU). The SB provides a

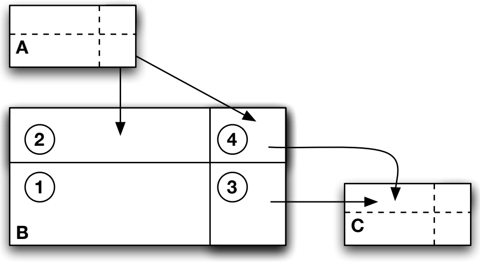


Figure 9.2: *The four categories of modules as defined within a Dependency Profile.* 1 = *hidden,* 2 = *inbound,* 3 = *outbound and* 4 = *transit modules. Arrows denote dependencies from/to modules within other components.*

value in the range of $[0,1]$ based on the number of components. The "optimal" number of components receives a high score which gradually decreases when a system contains a higher or a lower number of components. The "optimal" number of components is currently defined as the median number of components in a representative benchmark of systems.

The CSU metric provides a value in the range of $[0,1]$ based on whether the volume of the system is distributed (roughly) equally amongst the components of the system. More details about the design of these two metrics and their aggregation can be found in Chapter 6.

Component Independence To quantify Component Independence two categories of the Dependency Profiles are used. A dependency profile categorizes all modules within a component into one of four categories based on the dependencies the module has on modules in other components, as illustrated in Figure 9.2. The percentage of code within the system in the *internal* category and the *outbound* category is combined into a single percentage that quantifies the volume percentage of code in each component that is not called from or otherwise directly depended upon by code in another component. In other words, it quantifies the percentage of code which is not part of the *provided interface* of the components of the system. The higher this percentage the higher the rating for Component Independence.

Note that with this implementation the rating for Component Independence only focusses on the *provided interface* of the components of a system, the size of the *required interface* is not taken into account. The reasoning behind this is that the benefits of a small provided interface are widely known, and thus more likely to be accepted, than the need for a smaller required interface. However, in the tools used within SIG all four categories of the dependency profile are calculated and shown, thus allowing both interfaces to be discussed.

Implementation choices Similar to the existing system properties, the raw software metric is translated to a rating on a scale of $[0.5, 5.5]$ via a benchmarking based methodology (Alves et al., 2011), and these ratings are combined with the ratings for the other six system properties to calculate an overall rating (Baggen et al., 2010).

To ensure that both metrics can be applied to systems that are written in multiple programming languages the following algorithms are used. For Component Balance, the volume of the components is measured as the sum of Lines of Code for all programming languages used within a component.

To calculate the categories of the Dependency Profiles, language-specific hierarchy- and call-dependencies are statically resolved to calculate a percentage per programming language. The rating obtained per programming language is aggregated to the system-level using a weighted average, taking into account the relative volume of each language.

9.4.3 Consultants

In this research we observe the consultants working for SIG. These consultants provide recommendations to clients to improve the quality of a system in order to mitigate risks, but are not involved in the execution of these recommendations. Thus they fulfill the role of external quality assessors.

Two different types of consultants are distinguished, *technical* consultants trained in identifying and assessing technical risks within a software system and *general* consultants responsible for translating technical findings into project- and business risks. By observing both types of consultants we aim to gain more insight in both the technical usefulness as well as the usefulness of the metrics on a business level.

9.4.4 Services

Four different services are offered by SIG. Two of them, the Software Risk Assessment (SRA) and the Software Risk Monitor (SRM), are the main subjects of this research. The goal of an SRA is to answer a specific research question related to risks involved in the technical quality of a software system. A standard investigation lasts 6-8 weeks and is executed by a team consisting of both general and technical consultants (Deursen and Kuipers, 2003). In an SRM, the identified risks and technical quality of a system are tracked over a period of time to ensure timely identification and mitigation of problems (Kuipers and Visser, 2004).

9.4.5 Software subjects

In the six-month study period the measurement model has been used to monitor the technical quality of over 500 systems, and applied to over 50 systems in the setting of an SRA. The size of the systems varies from three thousand to several million lines of code written in a wide range of programming languages ranging from Object-Oriented and related languages (e.g., Java, C#, JSP, ASP, JavaScript and various SQL-dialects) to languages typically deployed in mainframes (e.g., Cobol, Tandem). The systems originate from different domains including banking, insurance, government and logistics.

To gain the most benefits from transferring to the new model a system needs to have its component defined. These definitions were made based on information retrieved from the development teams using interviews and design documents.

9.5 Data Gathering

After the metrics were embedded into the measurement model, data about the application and usage of the metrics was gathered using two different methods:

1. Experiences of using the metrics were collected through observations and documented in the form of memos

2. The opinions of the external assessors about the usefulness of the metrics were collected by conducting semi-structured interviews

Combining these two methods of data-gathering does not only allow us to triangulate findings, but also reduces the known limitations of either method. A reflection on this design decision is given in Section 9.9.

9.5.1 Observations

The objective of using this method is to gain an understanding of the challenges involved in applying the metrics, to gain insight in the situations in which the metrics can be used and to identify situations in which the values of the metrics do not directly correspond to the intuition of the assessor.

During the six month period the author of this thesis, who works as a technical consultants at SIG, collects experiences about the metrics in the form of written memos. All questions and remarks about the metrics which are either publicly stated or directly asked are documented on a daily basis.

Each memo contains a description of the problem/question, the answer provided/action defined and possible follow-up actions. After the six month period all memos are manually analyzed to identify recurring questions and general observations.

9.5.2 Interviews

The objective of using this method is to get an overview of the usefulness of the metrics as perceived by the external quality assessors. Eleven software quality assessors with at least two years of experience with performing metric-based quality assessments were interviewed to construct this overview.

The interviews are time-boxed to a period of 30 minutes and are conducted by the second author of the paper on which this chapter is based. This author, who is not involved in the daily operations of SIG, has previous experience in using interviews as a basis for qualitative research (Greiler et al., 2012) and started each interview with the following question:

> How do you use Component Balance and Component Independence?

The discussion based on this question is used to get a qualitative insight into the usefulness of the metrics. Each discussion is documented in a report which is validated by the interview subject.

In order to get a more quantitative insight into usefulness of the metrics each assessor was asked to answer the following two questions at the end of the interview:

1. On a scale of 1 to 5 (higher is better), how useful do you find Component Balance / Component Independence in your job?

2. On a scale of 1 to 5 (higher is better), does the use of Component Balance Component / Independence make it easier to do your job?

The above questions are based on the questions of Davis (1989) and are designed to get an insight into the perceived usefulness and ease of use of the metrics.

9.6 Observation Findings

Over the period of six months a total of 48 memos were collected. The memos describe interactions of the first author with 17 quality assessors (over one third of the available quality assessors), decomposed into 10 general consultants, six technical consultants and one internal researcher.

Twenty memos discuss specific systems, 14 different systems (spanning four different business contexts) where subject to these discussions. All memos combined involved 11 different customers and suppliers.

Note that even though the specific discussions only cover a fraction of all analyzed systems and clients, the remaining memos contain discussions about general trends observed by consultants, which are based on their experience with all systems as discussed in Section 9.4.5.

All memos were analyzed together at the end of the six months period. In this analysis 20 different concepts are extracted from the memos, which in turn are grouped into five different categories. Figure 9.3 shows an overview of the collected categories and the related concepts, each of which is discussed in the following sections.

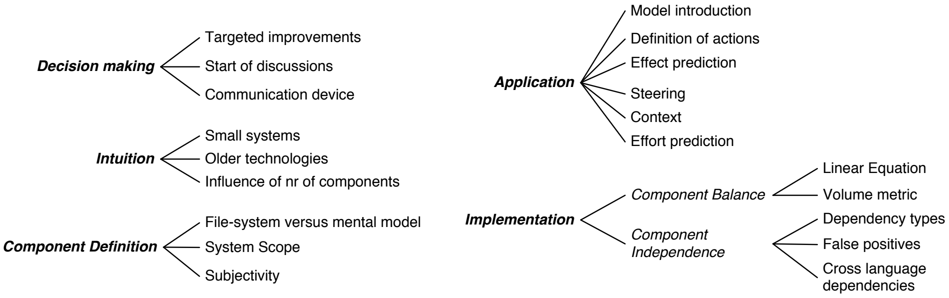


Figure 9.3: *An overview of the five categories (displayed in bold/italic font), two sub-categories (displayed in italic font) and their related concepts as collected via observations.*

9.6.1 Decision making

Indications for the actual use of the two architecture metrics in a decision making process was found in 16 different memos and grouped together in three different concepts.

Targeted Improvements The addition of the metrics to the measurement model resulted in targeted improvement efforts being made by development teams, including the development team employed by the company. Shortly after the introduction of the architecture metrics one of the internal developers posted the following message on the internal communications-channel:

> In ”eating-your-own-dogfood-news”, the new component independence metric helped us find a remnant of old design in [system-name] that was subsequently refactored, resulting in a +0.1 maintainability and a +0.85 component independence

Start of discussions Five memos describe direct questions posed by development teams on how to improve the rating for the metrics. In all cases, findings related to the specific system were discussed and recommendations were defined and communicated back to the development team.

Communication device One memo describes a discussion with a development team of a system which is being monitored. At first the monitoring was focussed on specific technical issues which required a technical componentization. However, these technical components did not correspond to the components used by the developers to reason about the system. In order to decrease the effort needed to transfer the system to a new maintenance team the current development team decided to re-structure the source-code to reflect their mental components. In this situation, both architecture metrics were used to communicate the progress of this re-structuring to project-management.

In summary, the metrics formed a basis for a discussion about the components, led to the definition of a roadmap to make the transition of the system easier and provided a way to track the progress for non-technical personnel.

9.6.2 Intuition

As explained by Fenton and Pfleeger (1998) a metric can be considered useful if it corresponds to the intuition of the person using the metric. On three occasions assessors specifically mention that in some cases the value of the metrics, in particular the Component Balance metric, does not immediately correspond to their intuition about the state of the system. One assessor states that in about half of the cases the ratings are as expected, while in the other half of the cases the definition of the components needs to be re-assessed.

More detailed examples of the situations in which the (change in) the value of the metrics does not correspond to the intuition of the assessors are described in seven different memos and grouped into three distinct categories.

Small systems Four different memos (involving five different assessors) describe that smaller systems seem to receive lower ratings for the architecture metrics faster than larger systems. One of the assessors hypothesized that this is due to smaller systems with components related to technical topics (typically only a few such as database, front-end, services) because the size of the system does not require a functional decomposition.

For one assessor this size-related issue was important enough to sit down together with an internal researcher to inspect the distribution of the ratings for all eight system properties to determine whether the distribution of ratings was indeed different. The result of this inspection was that, apart from a relatively large spike caused by systems which did not yet have component definition, the distribution of ratings for architecture metrics was not different from that of the other metrics.

Older technologies On two occasions assessors mentioned that systems written in older technologies (e.g., Cobol, Tandem, Pascal) seem to receive higher ratings for the component based metrics more easily than systems written in modern technologies (e.g., Java and C#).

For Component Balance, one assessor hypothesizes that this trend could be caused by the way in which components are solicited from the developers. Because the technologies themselves do not have a “component”-concept these types of systems normally do not have any components defined. During the transfer to the new quality model the sources are grouped together in components according to functionality *after* the metrics are explained, which could lead to a specific steering towards the “optimal” number of components and thus higher ratings.

Influence of number of components For Component Balance one assessor observes that the number of components seems to influence the rating for Component

Balance more than the size-distribution, which confirms the observations in the initial validation of this metric (see Chapter 6).

9.6.3 Component Definition

As discussed in Section 9.4, components needed to be defined for a system in order to gain the most from the transfer to the new model. Three concepts are related to this category.

File-system versus mental model The components of a system were defined based on either the structure of the file-system (e.g., the top-level directories are used as components) or based on interviews with developers about how they view the system. In the latter case it might well be that files from different directories are combined into a single component.

To illustrate, one assessor outlined a case in which the system contained a top-level directory structure depicting technical components, while the second-level directories contained a functional decomposition. Depending on the view-point of the developers either the functional or the technical components can be used to calculate the metrics. However, it was unclear to the assessor which one of the two is the best representation of the "real" components of the system and should thus be used for the current assessment.

Because there may be different components under various view-points, the value of the rating can diverge, which in turn can have political consequences (for example if there exists a contractual agreement to reach a certain rating for each system property). This type of situation calls for a more clear definition of what constitutes a component.

On the other hand, one assessor stated that within a SRA setting it can be helpful to use different component-definitions (representing different views on the system) to determine risks with respect to different view-points.

Subjectivity The lack of a very precise guideline of what constitutes a component is a reason to view a measurement based on components as subjective in two memos. In particular, by involving the developer in the definition of the components there exists a feeling that the value of the measurement can be easily influenced by using a different definition of component instead of a change in the code.

System Scope The question described in one memo was whether libraries developed inside the company (but in this case only included as a binary dependency) should be included as separate components. Even though this issue relates to the determination of system boundaries as opposed to the definition of component, it represents an important issue as it influences the number of components and thus the discussions based on the metrics. An additional challenge is that some technologies, e.g., JavaScript, enforce the source-code to be part of a code-base and thus influence the definition of the components (and other metrics).

9.6.4 Application

Challenges involved in the introduction and application of the metric are grouped together in six different concepts.

Model Introduction The transition to the new quality model has not been without challenges. After the initial introduction there was an additional need for an elevator pitch for the new metrics (requested in one memo). In addition, the investment needed to define components for a large number of systems written in older technologies, combined with viewing the definition of components as subjective, made one client decide not to upgrade yet to the new quality model for their portfolio.

Definition of Actions On three occasions different assessors indicate that defining actions based on the architecture metrics was more involved than providing advice for code-level metrics. On the code-level it is relatively straight-forward to define general actions (e.g., remove this type of duplication or refactor these long methods), but constructing this type of advice for the architecture metrics requires more effort because these recommendations are more context-dependent.

Effect prediction Related to the definition of actions, four memos describe that assessors are not always certain about the effect the implementation of a recommendation has on the value of the metrics. Especially because the addition of a component potentially affects both metrics, the results of implementing a recommendation is seen as harder to predict and could be smaller or bigger than desired.

For example, according to one assessor the effects of adding a component to a small system can significantly influence the overall rating, but one memo describes an example in which this influence was neglectable. Because of this uncertainty, assessors ask for tool-support to simulate the implementation of a recommendation in three different memos.

Steering One memo describes that the metrics are relatively stable for systems which are in maintenance mode. Any change in the metrics is normally the result of a targeted effort and thus expected. This makes it harder to use the metrics to steer development on a weekly basis, which is seen as disappointing.

Context As indicated in Section 9.6.1 suppliers directly ask for recommendations to increase the rating for a metric. In these situations it is tempting to focus on either one of the metrics to increase only that value. However, five memos describe discussions which point out that it is important to keep in mind that the eventual goal is not to have perfect ratings, but to have an architecture which fits the current needs.

For example, for one system the Component Independence received a low rating, while the rating for all other system properties are high. Combining this with the observation that the system has only one open issue at this time any effort spend to increase the rating for the this particular aspect is unneeded.

Effort prediction In line with the finding about providing recommendations, the effort prediction related to improving architectural issues is seen as difficult. Two memos describe that the effort needed to group components together is deemed to be lower than the effort required to split up components in separate chunks of functionality. In some situations a low rating is related to only a few violations, while in other situations a large refactoring is needed. Because of this the assessors experience difficulties in applying a general effort prediction model.

9.6.5 Implementation

Eight memos describe implementation issues regarding Component Balance (three) or Component Independence (five).

Component Balance Two memos (involving one assessor) describe a discussion about the implementation of the function to determine the rating for the number of components of a system. The conclusion of this discussion was that even though the function could be improved, the impact on the way in which the metric can be applied would be small. Secondly, the use of Lines of Code to depict the size of components is, according to one memo, not always applicable to XML-based languages.

Component Independence A first observation is that only using inter-language dependencies has a high impact on systems for which one component is implemented in a different technology. For example, in two systems one of the components was dedicated to an SQL-type language, while the other components used an Object-Oriented language. In these situations the rating for Component Independence was considered to be too high.

Secondly, two memos describe two different systems in which incorrect call resolving caused false positives, which in turn results in a rating which is too low (because modules are put into the wrong category).

Lastly, one memo outlines a conversation between an assessor and a supplier regarding the topic of dependency injection. As argued by the supplier, the difference in constructing a class directly or using a framework to construct and inject a class is small, but using the first approach significantly impacts the rating for Component Independence in a negative way.

9.7 Interview Findings

A total of eleven quality assessors were interviewed on three days over a period of one week. During the interviews notes were taken by the interviewer which formed the basis for a report of each interview. These reports have been validated by the subjects.

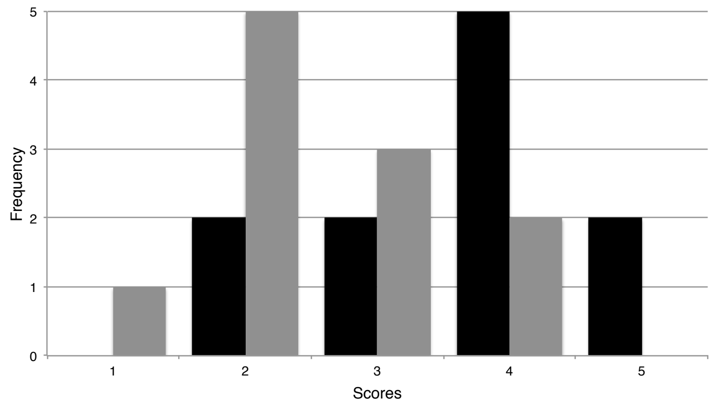


Figure 9.4: *Histogram displaying the distribution of scores related to the usefulness (black) and ease of use (gray) of the architecture metrics as given by the eleven subjects.*

The results of the interviews consists of two parts: a quantitative part based on the scores given by the subjects and a qualitative part in which the reports of the interviews are analyzed. The analyses of the reports has first been done by the authors on an individual basis, after which the results were discussed and combined. These results are presented in the remainder of this section based on the categories and concepts as displayed in Figure 9.3. Whenever a new finding could not be related to an existing concept, but was mentioned by at least two subjects, a new concept was introduced.

9.7.1 Subject scores

An overview of the scores given by the subjects is displayed in Figure 9.4. Scores related to the second question, whether the metrics are considered to be useful, are displayed in black, scores related to the third question, whether the metrics make their job easier, are displayed in gray.

From the distribution of scores we can deduce that overall the metrics are perceived to be useful, but that the application of the metrics does not make the job of assessments easier.

Related to the usefulness, nine subjects indicate that the metrics provide a starting point for discussions about the components of a system. In three interviews specific examples of how the metrics identified problems within a system were discussed, while in two interviews examples of the use of the metrics as a communication device were given. Lastly, three different subjects provide examples of targeted improvement efforts.

In relation to making the job of the consultant easier different types of challenges were brought forward, these challenges are discussed below.

9.7.2 Intuition

The concepts associated with this category were only discussed briefly. For example, only one subject mentions that smaller systems tend to score lower on the component metrics. And even though systems implemented in older technologies were discussed in four interviews, only one subject mentioned that they tend to score lower. Additional insights for this concept are discussed below. The seemingly large influence of the number of components on the rating was not discussed in any interview.

Older technologies One subject explained that older technologies tend to receive a lower rating, which was in-line with his expectation. Interestingly, another subject mentions that systems with older technologies normally receive a slightly better rating (an observation also made in the memos). From this we conclude that the exact influence of the architecture metrics on systems written in older technologies varies. Lastly, another subject mentions that for older technologies no meaningful componentization exists, thus the component metrics should play a less significant role in the overall assessment.

9.7.3 Component definition

The exact definition of components was a substantial topic in ten of the eleven interviews. In particular, each of the three concepts as described in the memos were mentioned by at least three subjects. Additional insights regarding these existing concepts are discussed below. In addition, the concept of *Technologies without components* is added to cover new findings in the interviews.

File-system versus mental model Seven subjects mention the challenge of choosing the “right” view on the components of a system. Apart from the mental model and the file-system, views related to the deployment of the code or functional decompositions can be chosen. A decision about which view should be leading in the calculation of the metrics is requested.

Subjectivity The fact that multiple view-points can be used to calculate the metrics leads to a feeling that the component definition is subjective. Although this flexibility is considered to be a good thing in the context of an SRA, where different view-points lead to different insights, the added flexibility sometimes leads to unwanted discussions about what constitutes a component (especially when the rating for the component metrics is low).

Technologies without components In three interviews it was mentioned that for some technologies the concept of a “component” does not exists. This does not only include older technologies such as Cobol and Tandem, but also systems implemented in newer technologies such as SAP. Additionally, visual programming languages, typically used to model business processes, also do not have an inherent concept of a component. It is yet unclear what the best way is to apply the component metrics to systems written in these technologies.

9.7.4 Application

With respect to the application of the metrics all concepts were mentioned in at least one interview. Several new insights were obtained for two different concepts. In addition, the concept of *Responsibility* is introduced to capture new findings.

Definition of actions Nine subjects mention that the definition of actions is harder for the architectural metrics than for the code-level metrics in the model. This is one of the main reasons why the new metrics do not make it easier to perform assessments. This increased effort needed to perform assessments is not necessarily seen as problematic. Definition of the actions might be harder, but is also seen as more interesting, challenging and valuable. However, to deal with common situations more efficiently one assessor suggests to collect experiences of applying the metrics to derive common recommendations.

Steering What is problematic is that in some situations the advice related to the components metrics can involve a substantial amount of refactoring. In some situations this leads to the identification of problems that cannot be solved due to resource constraints, which limits the usability of the advice in the setting of an SRA.

In a monitor setting the metrics can initially fluctuate, which is seen as problematic by one subject. However, two other subjects do not find this problematic as long as there is a roadmap towards a stable set of components. One subject mentions that within a monitoring setting it is easier to do something to improve the underlying architecture, although this is not necessarily done on a weekly basis.

Responsibility An additional challenge in the application of the metrics is that there exist situations in which the development team does not feel responsible for the components of the system. In some cases the components cannot be changed by the development team because the technology or framework dictates the component-structure; in other cases the components are mandated by a person outside the development team. In these situations a discussion about the value of the component metrics is considered to be useless by the development team, which hinders the application of the metrics.

9.7.5 Implementation

Ten interviews mentioned issues related to the implementation category. Issues related to the definition of Component Balance mainly revolved around a new concept of *Optimal number of components*; issues related to the other two concepts were not mentioned. With respect to Component Independence all three concepts were discussed at least once.

Component Balance - Optimal number of components As explained in Section 9.4.2 the “optimal” number of components is currently defined as the median number of components in a benchmark, which is currently 7. In five interviews this number is discussed, in all cases it is questioned whether the highest rating for SB should be attached to this number only.

9.8 Discussion of Findings

The data extracted from the 48 memos and the eleven interviews illustrate the usefulness of the Component Balance and Dependency Profiles metrics in the assessment of implemented architectures performed by external assessors. Examples show that the metrics can trigger targeted improvement efforts, start meaningful discussions and can be used as a communication device in a number of different situations.

The evaluation also illustrates situations in which the metrics perform less than optimal. For example, because the metrics are relatively stable for systems in maintenance mode they cannot be used to steer development on a weekly basis. Moreover, in some situations the recommendations following from the metrics require a significant amount of effort, which is not always available due to resource constraints.

In these situations the metrics illustrate a problem that is not subject to improvement, which reflects upon the perceived usefulness of the metrics.

Apart from these limitations, several areas of improvements are identified. We discuss those areas related to the three most discussed topics: Component Definition, Application and Intuition.

Component Definition The need for a strict definition of what entails a component across technologies is an important topic of discussion. Even though the ability to use different view-points is seen as both a positive and a negative aspect, a strict definition of component is asked for on several occasions. As illustrated in Section 9.6.1 there is a notion that the components of a system should be in line with the structure on the file-system, but the interviews indicate that such a definition is not applicable to all technologies.

In order to improve on this situation we plan to follow-up on the advice of one subject and collect a representative set of component definitions for different technologies. Such a set can be used as a basis for determining the components of existing systems, and provides an opportunity to derive general as well as technology-specific guidelines.

Application With respect to the application of the metrics both the observations and the interviews indicate that defining actions based on the value of the metrics is not always straight-forward. Moreover, estimating the amount of effort involved in implementing recommendations is considered challenging.

To deal with this problem we plan to build up a body of knowledge containing common value-patterns and associated recommendations for the architecture metrics. Having these common patterns and the effects of implementing the recommendations on the metrics available makes it easier for the assessors to gain a better feeling for the interpretation and application of the architecture metrics. The data collected in this evaluation should be considered a first step towards this body of knowledge.

Intuition According to the assessors, the value of the metric for smaller systems and systems written in specific technologies are not always as expected. Specific reasons for the mis-match between value and intuition seem to be the current determination of the "optimal" number of components for Component Balance and missing cross-language dependencies for Component Independence.

The overall intuition of the assessors could indicate that specific groups of systems should be treated differently by the metrics. To validate this hypothesis a statistical analyses on the values of the metrics in different groups could be performed, which is considered to be future work.

To address the specific issues we plan to investigate ways to implement support for cross-language dependencies in a cost-efficient manner. In addition, findings ways to make the resolving of dependencies more precise is deemed to be an important part of future work. For Component Balance we plan to investigate ways to better define the "optimal" number of components.

9.9 Reflections on Evaluation Methodology

In this section we reflect upon the benefits of evaluating the usefulness of metrics in practice and the used process.

First of all, the main benefit of performing this type of evaluation is a better understanding of the usefulness of the software metrics. Moreover, the effective identification of possible improvement areas illustrates the benefits of performing such an evaluation.

However, an important question here is whether the application of the metrics brings new insights, or whether all findings could have been defined before the evaluation. For example, one could argue that the need for a strict component definition or the questions related to systems written in different technologies could have been defined before the metrics were applied. However, even if this is true the relative importance of the different areas of future work could not have been determined in a purely academic context.

A second question related to the benefits of this type of evaluation is whether the results can be generalized to different contexts. In principle all findings are limited by the context as defined within Section 9.4. However, based on the depth of this evaluation and the nature of the metrics we expect that the benefits of the architecture metrics also apply to external assessors in different settings.

Note that it is important to be able to place the value of the metrics in a context, for example by the use of a benchmark. Because of this, the usefulness of the metrics for developers working on a single system is considered to be limited.

In relation to the followed methodology we make three important observations. First, the use of two different types of data-gathering is important to ensure a balanced evaluation. Secondly, the confidentiality constraints inherent to the evaluation of metrics within an industry setting limits reproducibility of the results. Lastly, the embedding of metrics within a standard operating procedure can be challenging.

Balanced Evaluation Every data-gathering technique has known limitations. For example, the interviews provide an indication of the usefulness of the metrics as perceived by the assessors. As pointed out by Davis (1989), perceived usefulness is not necessarily the same as objective usefulness. A limitation of the gathering of observations is the inherent confirmation bias of the observer.

These limitations are partially countered by combining the data from both methods. In addition, the two methods are executed by different persons to increase the possibility of finding new information in both methods. Furthermore, the interviews are conducted by the author that does not have daily interactions with the interview subjects to reduce interviewer bias.

The new findings in the interviews and the discovered areas of future work show that combining these two types of data-gathering leads to a balanced and critical evaluation of the usefulness of software metrics in practice.

Reproducibility Due to reasons of confidentiality, the data collected within the memos and the interviews cannot be made publicly available. However, the description of the data as given in Section 9.6 and Section 9.7 is considered to be detailed enough to support the conclusions drawn from the data.

Metric Embedding The biggest challenge in executing the proposed methodology is the embedding of new metrics in a standard operating procedure on a large scale, a topic that is out of scope of the current research. However, the benefits of evaluating metrics in practice as described above and in Section 9.8 is intended to assist researchers in acquiring the needed commitment from industry partners.

9.10 Related Work

Empirical evaluations of software metrics typically consist of evaluating the value of a metric against one of three external properties: quality in terms of faults, effort (either development or maintenance) or volume (Kitchenham, 2010).

By contrast, theoretical approached of metric evaluation inspect mathematical properties of metrics (Fenton and Pfleeger, 1998) or focus on metrological properties of metrics (Abran, 2010).

These types of evaluation aim to determine whether a metric is indeed measuring the attribute it was designed for in a theoretical manner. Kaner and Bond (2004) stress that this type of evaluation should also be done using a more practitioners oriented view-point and defines a framework for evaluating the validity and risk of a metric in the form of 10 questions.

All of the above evaluation strategies are meant to be done before a metric is used. Although useful, this pre-deployment validation covers only part of the 47 different validation criteria for metrics recently summarized in a literature review (Meneely et al., 2012). Our evaluation of the usefulness of software metrics bests fits the *actionability* criteria, which is defined as:

> A metric has actionability if it allows a software manager to make an empirically informed decision based on the software product's status (Meneely et al., 2012)

To the best of our knowledge, no empirical evaluation of this validation attribute has been done for specific metrics.

9.11 Conclusion

This chapter describes a large-scale industrial evaluation of the usefulness of the Component Balance and Dependency Profile metrics, in the context of the assessment of implemented architectures, from the view-point of external quality assessors. Using two different methods for gathering data, a detailed overview of the benefits and challenges of the two specific metrics is constructed and discussed.

For SIG, this evaluation identified different areas for improving the application of the metrics which have led to the definition of concrete improvement actions. For other practitioners, this evaluation can be used to decide whether or not the two architecture metrics can be used in their assessment processes. For the research community, the overview of areas for future work in Section 9.8, and the detailed overview of the data as discussed in Section 9.6 and Section 9.7, can be used as a starting point for conducting new research.

In addition, a methodology for evaluating software metrics in practice was introduced and the benefits and limitations of this approach are discussed. For practitioners, the overview of the insights gained from this type of evaluation is intended to inspire practitioners to collaborate with researchers to perform similar types of evaluations. For researchers, the methodology can serve as a starting point for evaluating the usefulness of (new) software metrics in practice, and can be reflected upon to improve the methodology itself.

To summarize, this chapter makes the following contributions:

- It introduces a methodology for evaluating the usefulness of software metric in industry
- It describes the execution of this methodology in an empirical study towards understanding the usefulness of two specific software metrics

- It provides an overview of challenges involved in the application of the two specific software metrics and lists concrete areas for improvement
- It reflects upon the usefulness of the evaluation methodology, concluding that the relative importance of challenges involved in applying specific metrics cannot be determined in a purely academic setting.

CHAPTER 10

Conclusion

Every software system has an implemented software architecture which allows or impedes the quality attributes of a system. It is therefore important to regularly evaluate this implemented architecture to ensure it is still suitable for the current context of the system. In this thesis we have investigated which aspects of an implemented software architecture should be evaluated with respect to maintainability, and provided concrete advice on how this evaluation can be performed by a quality evaluator. In particular, we have focussed on determining which metrics are able to quantify the modularity within a software system and evaluated the usefulness of these metrics within the evaluation of implemented architectures.

10.1 Contributions

The main contributions of this thesis can be summarized as follows:

- The identification of 15 system attributes that have an impact on the maintainability of an implemented architecture (Chapter 2), which have been incorporated in a general model for architecture complexity (Chapter 3).

- Practical tool- and process-support for starting evaluations in the form of the Lightweight Sanity Check for Implemented Architectures (Chapter 4) and the identification of four pitfalls related to using software metrics for evaluations (Chapter 5).

- A new metric to quantify the analyzability of a software architecture called Component Balance, including an evaluation of its construct validity (Chapter 6).

- A new metric to quantify the encapsulation of a software architecture called Dependency Profiles (Chapter 7), including an evaluation of its construct validity (Chapter 8).
- A qualitative evaluation of the usefulness of the Component Balance and the Dependency Profile metrics when used to evaluate the implemented architecture of software systems in practice (Chapter 9).

10.2 Answer to Research Questions

> RQ1: Which criteria should be taken into account during the evaluation of the maintainability of an implemented architecture?

The answer to this research question is given in Chapter 3, in the form of the Software Architecture Complexity Model (SACM) (see Figure 3.2). This model, which is partly based on the results presented in Chapter 2, consists of five factors, each of which are broken down into three to five criteria. Evaluating all criteria of the SACM leads to a balanced and rich overview of the complexity of an implemented architecture, which is related to the ease of future maintenance.

The five factors of SACM cover two different parts of complexity: the complexity of an architecture as experienced by a single developer (*personal* factors) and the complexity of the architecture as experienced by a team of developers (*environmental* factors). Depending on the criteria which receive a negative evaluation an initial estimate of the magnitude of the solution can be defined. For example, changing the technology in which a systems is written (an *environmental* factor) will normally require more effort than removing a cyclic dependency (a *personal* factor). Thus the SACM can not only be used as a check-list for evaluations, but also provides a structured way of reasoning about solutions for solving the complexity of an implemented architecture.

The SACM is based on well-accepted theories taken from the field of cognitive science and general software engineering principles. In addition, the criteria of SACM correspond to the criteria used by quality evaluators to evaluate implemented architectures of systems written in a wide range of technologies. Despite the limitations of the model as discussed in Section 3.6.7, in particular the fact that it is unknown whether the model is complete, SACM provides a solid, generic basis for the evaluation of the implemented architecture of software systems.

> RQ2: What support can we define to make the process of regularly checking an implemented architecture easier for a quality evaluator?

The answer to this research question is two-fold. First, Chapter 4 introduces the Lightweight Sanity Check for Implemented Architecture (LiSCIA), consisting of 28 questions within five different categories. The design of LiSCIA is such that the effort needed to apply LiSCIA for a person familiar with the system under evaluation, or an experienced quality evaluator, is less than one day. Apart from listing the questions, LiSCIA also provides possible actions for improvements. Because of this LiSCIA provides quality evaluators with a ready-to-use check-list to start performing evaluations.

Secondly, Chapter 5 explains four common pitfalls related to the use of (software) metrics within project management and quality evaluations. These pitfalls are structured around two aspects of metric usage: the *meaning* given to metric values and the *number* of metrics used. For both categories we have defined names and descriptions for situations in which either too much or too little of the aspect is involved. By taking into account these pitfalls quality evaluators and project managers can achieve their goals more effectively.

The biggest limitation of the answer to this question is the lack of rigorous evaluation of both LiSCIA and the four pitfalls. The results of an initial evaluation of LiSCIA, in which the check-list was embedded in the quality assurance process of the services of SIG, showed that the check-list was a nuisance for experienced quality evaluators, e.g., filling in the check-list was seen as a 'red-tape' activity instead of being helpful in the evaluation process. However, a formal evaluation of LiSCIA being applied by novice quality evaluators is yet to be done.

Likewise, the evaluation of whether it is useful to know the four pitfalls when applying metrics in a project management situation is an open research topic. Even though anecdotical evidence shows that knowing the pitfalls helps to apply metrics more effectively, a formal experiment to test this hypothesis is an open research task.

> RQ3: Which metrics are capable of quantifying the modularization of a software system?

To answer this question we identified two metrics to quantify two different aspects of the modularization of a system; the analyzability of a system in terms of its components and the encapsulation of a system in terms of its dependencies.

The experiment within Chapter 6 shows that Component Balance is the most promising metric to quantify the analyzability of a software system in terms of its components. This metric is the first metric which combines both the number of components and their relative sizes, both of which are important criteria according to quality evaluators. When a system consists of a reasonable number of components which are roughly equal in size the value of the metric will be higher, indicating a better analyzability.

The experiment within Chapter 8 shows that the different categories of the Dependency Profiles, in particular the amount of *internal* code, are the most promising metrics to quantify the encapsulation within a system based on the dependencies between components. When the percentage of internal code, i.e., code which only uses and is used by code within a single component, is higher, the encapsulation within a system is considered to be better.

For both metrics we have used controlled experiments to validate their construct validity. By including existing architecture level metrics designed to quantify the same characteristic within each experiment we have ensured that these new metrics out-perform the current state of the art metrics which can be calculated on the same input.

RQ4: Are the metrics identified in RQ3 useful in practice?

The empirical validation as presented in Chapter 9 shows that the Component Balance and the different categories of the Dependency Profile are considered useful, in particular when deployed using a benchmark-based approach as explained in Section 9.4.2.

The validation zoomed in on two aspects which indicate the usefulness of a metric: the correspondence of the value of the metric to the intuition of the quality evaluator and the usage of the value of the metric in a decision making process. In relation to the first aspect the collected data shows that metrics tend to correspond to the intuition of the quality evaluator, or at least form a basis for a constructive discussion of the implemented architecture of a system. The results of these discussions lead to the definition of actions, which shows that the metrics are indeed used for making decisions.

The empirical validation also resulted in the definition of several areas of improvements related to, amongst others, the specific metrics, tool support for architecture evaluations and the way in which metric-based evaluations can be improved in general. For all of these areas we define directions for future work.

Note that this answer is only one of the possible answers, different (combinations of) architecture-level metrics might be as useful (or even more useful) than the combination of Component Balance and the Dependency Profile. However, without a validation similar to the one presented in Chapter 9 we cannot draw any conclusions regarding the usefulness of other (combinations of) architecture-level metrics. Fortunately, the evaluation design as presented in Chapter 9 is metric agnostic and can be reused with little effort.

10.3 Impact on Practice

For our specific research context of SIG, this research has enabled a structural change regarding the evaluation of implemented software architectures. As explained in Section 9.4.1, SIG used to deploy a quality model based on ISO/IEC 9126 (International Organization for Standardization, 2001) as a basis for its consultancy services (Baggen et al., 2010).

Based on, amongst others, the results described in this thesis this model has been updated to reflect the changes in ISO/IEC 25010 (International Organization for Standardization, 2011), which is the successor of ISO/IEC 9126. In this update, the quality model is augmented with two new system properties targeted towards quantifying aspects related to the modularity of the system. The metrics used to quantify these system properties are based on Component Balance and the Dependency Profile (see Section 9.4.2 for more details).

This means that the metrics as proposed and evaluated in this thesis are used to continuously monitor the quality of over 500 software systems, and are used in structured evaluations of over 100 systems on a yearly basis. Moreover, the metrics are used in the measurement procedure approved by TÜViT for the "Trusted Product Maintainability"-certificates[1] since the beginning of 2012.

Overall, the embedding of these metrics inside the standard software quality model used to evaluate the maintainability of software systems has lead to a more structured approach towards evaluating implemented software architectures.

10.4 Impact on Research

Apart from the contributions and the practical impact this research advances the field of software engineering research in three different ways.

First of all, we have defined and executed an evaluation methodology which enables a meaningful interpretation of studies which correlate the value of a snapshot-based metric and the value of a metric which is calculated over a period of time (see Chapter 8). A crucial part of this methodology is the acknowledgment that the value of the snapshot-based metric must be representative for a given period of time in order to ensure that conclusions drawn from the correlation numbers are correct. Any study in which the correlation between these two types of metrics is investigated needs to take this consideration into account.

Secondly, we have defined and executed evaluation methodology to valid the usefulness of a combination of software metrics in practice (see Chapter 9). By showing the value of this type of studies it becomes easier for other researchers to convince industrial partners to cooperate in this type of evaluation.

Lastly, this thesis is an example of the "industry-as-a-laboratory" approach as sketched out by Potts (1993). We first identified a problem in our particular research context and then used data from that research context to define a solution. This solution is in turn validated by experiments designed using well-established research methods, after which the solution is applied again in practice. The combination of research methods and constant interaction between research and industry has resulted in generally applicable, yet immediately useful, solutions. Based on the success of this research we hope to inspire other researchers and practitioners to adopt the same methodology.

[1] `https://www.tuvit.de/en/products/maintainability-1215.htm`

10.5 Future work

We envision four areas of future work, which roughly follow the answer to our research questions in reverse order from smaller to larger research areas.

10.5.1 Improving specific metrics

With respect to Component Balance and the Dependency Profile we have identified several specific tracks for future work in Section 9.8. For Component Balance we first plan to examine the influence of the number of components on the overall score in more detail. Furthermore, the inspection of the (difference in) statistical behavior of the metrics with respect to systems written in different technologies is an important short-term research topic.

With respect to the usage of the metrics the collection of common usage patterns is deemed useful. For example, we envision the construction a "metric-interpretation" catalog in which these patterns are collected together with the recommendations made and the required effort for implementing the recommendations (if any). Such a catalog provides practitioners with hands-one advice, while researchers can use the patterns as a source of inspiration for conducting research.

Note that the construction of such a catalog is not limited to the Component Balance and the Dependency Profile metrics, every software metric can benefit from such a catalog.

10.5.2 Improving criteria evaluation

Concerning the evaluation of other criteria as defined in SACM we see a wide range of potential research topics. The research described in this thesis designed and validated metrics for two specific criteria: *balance* and *independence*. For the remaining sixteen criteria similar research projects can be executed to define metrics or investigate automation techniques. This would enable a more efficient evaluation of these remaining criteria as opposed to only using expert opinion.

For example, one of the inputs for evaluating the criteria of *Inner Coherence* is the set of responsibilities implemented within modules. Unfortunately, for most systems the mapping between responsibilities/functionality and source-code modules is unclear. The effort involved in constructing and maintaining this mapping is (at the current time) too large of an investment. To somewhat lift this limitations we have been working towards a more automated way in towards the construction of such a mapping (Olszak et al., 2012).

10.5.3 Improving characteristic evaluation

Taking another step back we emphasize that this research covers (a part of) only one of the eight sub-characteristics of the ISO/IEC 25010 model for software quality.

For each of the remaining seven quality characteristics we see possibilities for future research.

In general, each quality characteristic can benefit from the definition of a structured evaluation approach. As soon as there is a considerable amount of data collected through the execution of such a structured approach a quality model can be developed based on this data and the experience of quality evaluators.

10.5.4 Improving metric based evaluations

Lastly, we see the identification of the four pitfalls of using software metrics as defined in Chapter 5 as a first step towards the identification of the limits which are involved in using metrics in an evaluation setting. Metrics are a powerful tool to identify possible areas of improvements, but are aware that the wrongful use of metrics can do more harm than good.

To prevent this harm from happening metrics should always be (correctly) interpreted by humans. Developing support and education materials to enable the correct interpretation of metrics is a significant area of future work. This support is a first stepping stone towards the development of a truly fact-based software engineering community, ultimately leading to the development of better and more accurate software systems.

Bibliography

A. Abran. *Software Metrics and Software Metrology*. Wiley-IEEE Computer Society Press, 2010.

S. Adolph, W. Hall, and P. Kruchten. Using grounded theory to study the experience of software development. *Empirical Software Engineering*, 16(4):487–513, August 2011.

A.J. Albrecht and Jr. Gaffney, J.E. Software function, source lines of code, and development effort prediction: A software science validation. *IEEE Transactions on Software Engineering*, SE-9(6):639 – 648, 1983.

E.B. Allen and T.M. Khoshgoftaar. Measuring coupling and cohesion: An information-theory approach. In *Proceedings of the 6th International Symposium on Software Metrics*. IEEE Computer Society, 1999.

M. AlSharif, W.P. Bond, and T. Al-Otaiby. Assessing the complexity of software architecture. In *ACM-SE 42: Proceedings of the 42nd annual Southeast regional conference*, pages 98–103. ACM, 2004.

T.L. Alves, J.P. Correia, and J. Visser. Benchmark-based aggregation of metrics to ratings. In *Proceedings of the Joint Conference of the International Workshop on Software Measurement and the International Conference on Software Process and Product Measurement (IWSM/MENSURA)*, pages 20–29, 2011.

M. Anan, H. Saiedian, and J. Ryoo. An architecture-centric software maintainability assessment using information theory. *Journal of Software Maintenance and Evolution*, 21:1–18, January 2009.

J.R. Anderson. *Cognitive psychology and its implications*. W.H.Freeman & Co Ltd, 2000.

A. Avritzer and E.J. Weyuker. Investigating metrics for architectural assessment. In *METRICS '98: Proceedings of the 5th International Symposium on Software Metrics*, page 4. IEEE Computer Society, 1998.

M.A. Babar and I. Gorton. Software architecture review: The state of practice. *Computer*, 42(7):26–32, 2009.

M.A. Babar, L. Zhu, and D.R. Jeffery. A framework for classifying and comparing software architecture evaluation methods. In *ASWEC '04: Proceedings of the 2004 Australian Software Engineering Conference*, page 309. IEEE Computer Society, 2004.

R. Baggen, K. Schill, and J. Visser. Standardized code quality benchmarking for improving software maintainability. In *4th International Workshop on Software Quality and Maintainability (SQM 2010)*, 2010.

V.R. Basili, G. Caldiera, and H.D. Rombach. The goal question metric approach. In *Encyclopedia of Software Engineering*. Wiley, 1994.

L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice, Second Edition*. Addison-Wesley Professional, 2003.

G. Beliakov, A. Pradera, and T. Calvo. *Aggregation Functions: A Guide for Practitioners*. Springer Publishing Company, Incorporated, 2008.

J.K. Blundell, M.L. Hines, and J. Stach. The measurement of software design quality. *Annals of Software Engineering*, 4(1022-7091):235–255, 1997.

G. Booch. *Object-oriented analysis and design with applications (2nd ed.)*. Benjamin-Cummings Publishing Co., Inc., 1994.

E. Bouwers and A. van Deursen. A lightweight sanity check for implemented architectures. *IEEE Software*, 27(4), 2010.

E. Bouwers, J. Visser, and A. van Deursen. Criteria for the evaluation of implemented architectures. In *Proceedings of the 25th International Conference on Software Maintenance (ICSM 2009)*, pages 73–82. IEEE Computer Society, 2009.

E. Bouwers, C. Lilienthal, J. Visser, and A. van Deursen. A cognitive model for software architecture complexity. In *Proceedings of the 2010 IEEE 18th International Conference on Program Comprehension*. IEEE Computer Society, 2010.

E. Bouwers, J.P. Correia, A. van Deursen, and J. Visser. Quantifying the analyzability of software architectures. In *Proceedings of the 9th Working IEEE/IFIP Conference on Software Architecture (WICSA 2011)*. IEEE Computer Society, 2011a.

E. Bouwers, A. van Deursen, and J. Visser. Quantifying the encapsulation of implemented software architectures. Technical Report TUD-SERG-2011-031, Delft Software Engineering Research Group, Delft University of Technology, 2011b.

E. Bouwers, A. van Deursen, and J. Visser. Dependency profiles for software architecture evaluations. In *Proceedings of the 27th IEEE International Conference on Software Maintenance (ICSM 2011)*. IEEE Computer Society, 2011c.

E. Bouwers, J. Visser, and A. van Deursen. Getting what you measure. *Communications of the ACM*, 55(7):54–59, July 2012.

E. Bouwers, A. van Deursen, and J. Visser. Evaluating usefulness of software metrics: An industrial experience report. In *Proceedings of the 35th International Conference on Software Engineering (ICSE 2013)*, 2013.

L.C. Briand, S. Morasca, and V.R. Basili. Measuring and assessing maintainability at the end of high level design. In *Proceedings of the Conference on Software Maintenance (ICSM 1993)*, pages 88–97. IEEE Computer Society, 1993.

L.C. Briand, J.W. Daly, and J.K. Wust. A unified framework for coupling measurement in object-oriented systems. *IEEE Transactions of Software Engineering*, 25 (1):91 –121, jan/feb 1999a.

L.C. Briand, S. Morasca, and V.R. Basili. Defining and validating measures for object-based high-level design. *IEEE Transactions on Software Engineering*, 25 (5):722–743, 1999b.

B. Brügge and A.H. Dutoit. *Object oriented Software Engineering Using UML, Patterns, and Java*. Prentice Hall International, 2009.

S.R. Chidamber and C.F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20:476–493, 1994.

H.B. Christensen, K.M. Hansen, and B. Lindstrøm. Lightweight and continuous architectural software quality assurance using the asqa technique. In *Proceedings of the 4th European conference on Software architecture*, ECSA'10, pages 118–132. Springer-Verlag, 2010.

P. Clements, R. Kazman, and M. Klein. *Evaluating software architectures : methods and case studies*. Addison-Wesley, 2002.

P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord, and J. Stafford. *Documenting Software Architectures: Views and Beyond*. Addison-Wesley, 2003.

A. Cockburn. *People and Methodologies in Software Development*. PhD dissertation, University of Oslo, 2003.

J.P. Correia and J. Visser. Certification of technical quality of software products. In *Proceedings of the 2nd International Workshop on Foundations and Techniques for Open Source Software Certification*, pages 35–51, 2008.

L. Couto, J.N. Oliveira, M. Ferreira, and E. Bouwers. Preparing for a literature survey of software architecture using formal concept analysis. In *Proceedings of the Fifth International Workshop on Software Quality and Maintainability (SQM 2011)*, 2011.

J.W. Creswell and V.L.P. Clark. *Designing and Conducting Mixed Methods Research*. Sage Publications, Inc, 1 edition, August 2006.

F.D. Davis. Perceived usefulness, perceived ease of use, and user acceptance of information technology. *MIS Quarterly*, 13(3):319–340, 1989.

A. van Deursen and T. Kuipers. Source-based software risk assessment. In *ICSM '03: Proceedings of the International Conference on Software Maintenance*. IEEE Computer Society, 2003.

L. Dobrica and E. Niemelä. A survey on software architecture analysis methods. *IEEE Transactions on Software Engineering*, 28(7):638–653, 2002.

C. Ebert. Complexity traces: an instrument for software project management. pages 166–176. International Thomson Computer Press, 1995.

E. Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley Professional, August 2003.

N.E. Fenton and S.L. Pfleeger. *Software Metrics: A Rigorous and Practical Approach*. PWS Publishing Co., 2nd edition, 1998.

M. Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley Longman Publishing Co., Inc., 2002.

E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.

A. Gawande. *The Checklist Manifesto: How to Get Things Right*. Metropolitan Books, December 2009.

C. Gini. Measurement of inequality of income. *Economic Journal*, 31:22–43, 1921.

A. Gopal, T. Mukhopadhyay, and M.S. Krishnan. The impact of institutional forces on software metrics programs. *IEEE Transactions on Software Engineering*, 31 (8):679 – 694, aug. 2005.

T.L. Graves, A.F. Karr, J.S. Marron, and H. Siy. Predicting fault incidence using software change history. *IEEE Transactions on Software Engineering*, 26:653–661, July 2000.

M. Greiler, A. van Deursen, and M. Storey. Test confessions: a study of testing practices for plug-in systems. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 244–254. IEEE Press, 2012.

G. Gui and P. Scott. Ranking reusability of software components using coupling metrics. *Journal of Systems and Software*, 80(9):1450–1459, September 2007.

M. Haft, B. Humm, and J. Siedersleben. The architect's dilemma - will reference architectures help? In R. et al. Reussner, editor, *Quality of Software Architectures and Software Quality QoSA/SOQUA*, pages 106–122, 2005.

N. Harrison and P. Avgeriou. Pattern-based architecture reviews. *IEEE Software*, PP (99):1, 2010.

L. Hatton. Reexamining the fault density-component size connection. *IEEE Software*, 14(2):89–97, 1997.

I. Heitlager, T. Kuipers, and J. Visser. A practical model for measuring maintainability. In *QUATIC '07: Proceedings of the 6th International Conference on Quality of Information and Communications Technology*, pages 30–39. IEEE Computer Society, 2007.

B. Henderson-Sellers. *Object-oriented metrics: measures of complexity*. Prentice-Hall, Inc., 1996.

W.G. Hopkins. *A new view of statistics.* Internet Society for Sport Science, 2000. URL `http://www.sportsci.org/resource/stats/`.

D.H. Hutchens and V.R. Basili. System structure analysis: Clustering with data bindings. *IEEE Transactions on Software Engineering*, 11(8):749–757, 1985.

E. Hutchins. *Cognition in the wild.* MIT Press, 1996.

IEEE. IEEE Std 610.12-1990: IEEE standard glossary of software engineering terminology, 1990.

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

International Organization for Standardization. ISO/IEC 25010: Systems and software engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) - System and software quality models, 2011.

C. Kaner and W.P. Bond. Software engineering metrics: What do they measure and how do we know? In *10th International Software Metrics Symposium - Metrics 2004*. IEEE Computer Society Press, 2004.

R. Kazman and M. Burth. Assessing architectural complexity. In *CSMR '98: Proceedings of the 2nd Euromicro Conference on Software Maintenance and Reengineering (CSMR'98)*, page 104. IEEE Computer Society, 1998.

R. Kazman and S.J. Carrière. Playing detective: Reconstructing software architecture from available evidence. *Automated Software Engineering*, 6(2), 1999.

B. Kitchenham. Whats up with software metrics? A preliminary mapping study. *Journal of Systems and Software*, 83(1):37 – 51, 2010.

P. Kogut and P. Clements. The software architecture renaissance. *Crosstalk - The Journal of Defense Software Engineering*, 7:20–24, 1994.

H. Koziolek. Sustainability evaluation of software architectures: a systematic review. In *Proceedings of the joint ACM SIGSOFT conference – QoSA and ACM SIGSOFT symposium – ISARCS on Quality of software architectures – QoSA and architecting critical systems – ISARCS*, QoSA-ISARCS '11, pages 3–12. ACM, 2011.

P. Kruchten. The 4+1 view model of architecture. *IEEE Software*, 12(6):42–50, 1995.

T. Kuipers and J. Visser. A tool-based methodology for software portfolio monitoring. In *Proceedings of the 1st International Workshop on Software Audit and Metrics*, pages 118–128. INSTICC Press., 2004.

J. Lakos. *Large-scale C++ software design*. Addison Wesley Longman Publishing Co., Inc., 1996.

M. Lehman. On understanding laws, evolution and conservation in the large program life cycle. *Journal of Systems and Software*, 1(3):213–221, 1980.

C. Lilienthal. *Komplexität von Softwarearchitekturen, Stile und Strategien*. PhD dissertation, Universität Hamburg, Software Engineering Group, 2008.

C. Lilienthal. Architectural complexity of large-scale software systems. In *Proceedings of the 13th European Conference on Software Maintenance and Reengineering*. IEEE Computer Society Press, 2009.

M. Lindvall, R. Tesoriero Tvedt, and P. Costa. An empirically-based process for software architecture evaluation. *Empirical Software Engineering*, 8(1):83–108, 2003.

H. Lu, Y. Zhou, B. Xu, H. Leung, and L. Chen. The ability of object-oriented metrics to predict change-proneness: a meta-analysis. *Empirical Software Engineering*, 17 (3):1–43, 2012.

S. Mancoridis, B.S. Mitchell, C. Rorres, Y. Chen, and E.R. Gansner. Using automatic clustering to produce high-level system organizations of source code. In *IWPC '98: Proceedings of the 6th International Workshop on Program Comprehension*. IEEE, 1998.

S. Mancoridis, B.S. Mitchell, Y. Chen, and E.R. Gansner. Bunch: A clustering tool for the recovery and maintenance of software system structures. In *Proceedings of the IEEE International Conference on Software Maintenance*, ICSM '99. IEEE Computer Society, 1999.

R.C. Martin. *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall, 2002.

T. McCabe and C.W. Butler. Design complexity measurement and testing. *Communications of the ACM*, 32(12):1415–1425, 1989.

T.J. McCabe. A complexity measure. In *ICSE '76: Proceedings of the 2nd international conference on Software engineering*. IEEE Computer Society Press, 1976.

J.A. McCall, P.K. Richards, and G.F. Walters. Factors in software quality. In *US Rome Air Development Center*, pages Nr. RADC TR–77–369, Vols I,II,III, 1977.

J.A. McDermid. Complexity: concept, causes and control. In *Proceedings of the Sixth IEEE International Conference on Engineering of Complex Computer Systems (ICECCS), 2000.*, pages 2–9, 2000.

N. Medvidovic and V. Jakobac. Using software evolution to focus architectural recovery. *Automated Software Engineering*, 13(2):225–256, 2006.

H. Melton and E. Tempero. An empirical study of cycles among classes in java. *Empirical Software Engineering*, 12(4):389–415, 2007.

A. Meneely, B. Smith, and L. Williams. Validating software metrics: A spectrum of philosophies. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 21, 2012.

G. Miller. The magical number seven, plus or minus two: Some limits on our capacity for processing information. *The Psychological Review*, 63(2):81–97, 1956.

G.C. Murphy, D. Notkin, and K.J. Sullivan. Software reflexion models: Bridging the gap between source and high-level models. In *SIGSOFT '95: Proceedings of the 3rd ACM SIGSOFT symposium on Foundations of software engineering*, pages 18–28. ACM, 1995.

N. Nagappan and T. Ball. Static analysis tools as early indicators of pre-release defect density. In *Proceedings of the 27th international conference on Software engineering*, ICSE '05, pages 580–586. ACM, 2005.

D.A. Norman. *Learning and Memory*. W. H. Freeman & Co, ACM Press, 1982.

A. Olszak, E. Bouwers, B.N. Jørgensen, and J. Visser. Detection of seed methods for quantification of feature confinement. In *TOOLS Europe 2012: Proceedings of the 50th 50th International Conference on Objects, Models, Components, Patterns*, 2012.

D.L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, 1972.

D.E. Perry and A.L. Wolf. Foundations for the study of software architecture. *SIGSOFT Software Engineering Notes*, 17(4):40–52, 1992.

C. Potts. Software-engineering research revisited. *IEEE Software*, 10(5):19–28, September 1993.

F. Reichheld. The one number you need to grow. *Harvard business review*, 81(12): 46–54, 2003.

A.J. Riel. *Object-Oriented Design Heuristics*. Addison-Wesley, April 1996.

D. Romano and M. Pinzger. Using source code metrics to predict change-prone java interfaces. In *Proceedings of the 27th IEEE International Conference on Software Maintenance (ICSM 2011)*. IEEE Computer Society, 2011.

D. Romano, M. Pinzger, and E. Bouwers. Extracting dynamic dependencies between web services using vector clocks. In *Proceedings of the 2011 IEEE International Conference on Service-Oriented Computing and Applications*, SOCA '11, pages 1–8. IEEE Computer Society, 2011.

R.S. Sangwan, P. Vercellone-Smith, and P.A. Laplante. Structural Epochs in the complexity of Software over Time. *IEEE Software*, 25(4):66–73, 2008.

C. Sant'Anna, E. Figueiredo, A. Garcia, and C. Lucena. On the modularity of software architectures: A concern-driven measurement framework. In Flavio Oquendo, editor, *Software Architecture*, volume 4758 of *Lecture Notes in Computer Science*, pages 207–224. Springer Berlin / Heidelberg, 2007.

S. Sarkar, G.M. Rama, and A.C. Kak. API-based and information-theoretic metrics for measuring the quality of software modularization. *IEEE Transactions of Software Engineering*, 33(1):14–32, 2007.

S. Sarkar, A.C. Kak, and G.M. Rama. Metrics for measuring the quality of modularization of large-scale object-oriented software. *IEEE Transactions on Software Engineering*, 34:700–720, 2008.

S.S. Shapiro and M.B. Wilk. An analysis of variance test for normality (complete samples). *Biometrika*, 52(3-4):591–611, 1965.

H.A. Simon. *The Sciences of the Artificial*. MIT Press, 1996.

W.P. Stevens, G.J. Myers, and L.L. Constantine. Structured design. *IBM Systems Journal*, 13(2):115–139, 1974.

M.-A.D. Storey, F.D. Fracchia, and H.A. Müller. Cognitive design elements to support the construction of a mental model during software exploration. *Journal of Systems and Software*, 44(3):171–185, 1999.

M. Svahnberg. *Supporting Software Architecture Evolution*. PhD thesis, Blekinge Institute of Technology, 2003.

P. Tarr, H. Ossher, W. Harrison, and S.M. Sutton, Jr. N degrees of separation: multi-dimensional separation of concerns. In *ICSE '99: Proceedings of the 21st international conference on Software engineering*, pages 107–119. ACM, 1999.

J. van Gurp and J. Bosch. Design erosion: problems and causes. *Journal of Systems and Software*, 61(2):105–119, 2002.

R. Vasa, M. Lumpe, P. Branch, and O. Nierstrasz. Comparative analysis of evolving software systems using the gini coefficient. In *Proceedings of the 25th International Conference on Software Maintenance (ICSM)*, pages 179–188. IEEE, 2009.

C. Wohlin, P. Runeson, M. Host, C. Ohlsson, B. Regnell, and A. Wesslén. *Experimentation in Software Engineering: an Introduction*. Kluver Academic Publishers, 2000.

R.K. Yin. *Case study research: Design and methods*, volume 5. Sage Publications, Inc, 2009.

A.T.T. Ying, G.C. Murphy, R. Ng, and M.C. Chu-Carroll. Predicting source code changes by mining change history. *IEEE Transactions on Software Engineering*, 30:574–586, September 2004.

E. Yourdon and L. L. Constantine. *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*. Prentice-Hall, Inc., 1979.

L. Yu, A. Mishra, and S. Ramaswamy. Component co-evolution and component dependency: speculations and verifications. *IET Software*, 4(4):252–267, 2010.

J. Zhao. On assessing the complexity of software architectures. In *ISAW '98: Proceedings of the third international workshop on Software architecture*, pages 163–166. ACM, 1998.

T. Zimmermann, S. Diehl, and A. Zeller. How history justifies system architecture (or not). In *Proceedings of the 6th International Workshop on Principles of Software Evolution*, pages 73–83. IEEE Computer Society, 2003.

H. Zuse. *Software Complexity: Measures and Methods*. Walter de Gruyter & Co., 1990.

Summary

Software systems make up an important part of our daily lives. Just like all man-made objects, the possibilities of a software system are constrained by the choices made during its creation. The complete set of these choices can be referred to as the *software architecture* of a system.

Since the software architecture of a system has a large influence on what can, and cannot, be done with the system, it is important to regularly evaluate this architecture. The purpose of such an evaluation is to create an overview of the strengths and weaknesses of the software architecture, which can then be used to decide whether each weakness is accepted or should be addressed.

There is a wide range of software architecture evaluation methods available which can be used to investigate one or more quality aspects of a software architecture. Most of these methods focus on the initial *design* of the software architecture, there are only a few which specifically target the *implemented* architecture.

Looking at the design alone is not problematic if design and implementation are in sync, but unfortunately there are many occasions in which these two architectures deviate. Moreover, some systems are even built without an initial design at all. In addition, earlier research shows that software architectures are not regularly evaluated in practice, despite the availability of these methods. The reason for this is that the initial effort to start performing software architecture evaluations is considered to be too high for project teams.

The goal of this thesis is to lower this initial investment by providing an overview of concrete evaluation attributes, as well as the definition of software metrics that can be used to evaluate these attributes.

Our global research approach is that of “industry-as-a-lab”. During our research we have closely collaborated with the Software Improvement Group to design solutions, and to test these solutions on real-world cases. To be able to provide concrete advice we must focus on only a single quality attribute; in this thesis we choose to focus on the *maintainability* quality attribute of a software system.

Following from our goal the research in this thesis is composed of two parts. The focus of the first part is on the identification of architectural attributes that can be used as an indicator for the maintainability of a software system. The result of mining over 40 evaluation reports, interviewing experts, and a validation with various experts is a list of 15 architectural attributes which experts consider to be indicators for the maintainability of a software system.

To augment the opinion of the experts we used theories from the field of cognitive psychology to extend an existing model for architectural complexity. This extended model makes it possible to explain *why* each of the found attributes influence the maintainability of a software system.

Based on the attributes and the model we developed a lightweight sanity-check for implemented architectures. This check consists of 28 questions and 28 actions divided over five categories. A person familiar with a system can use this check to get an initial overview of the status of their system within a day, and needs less effort to repeat this evaluation later on.

In the second part of our research we focus on the design and validation of metrics related to two quality attributes: *balance* and *independence*. These two attributes are related to two of the major building blocks of an implemented architecture; the definition of the components of the system and the relationship these components have with each other. In the ideal case a system is decomposed into a limited set of components on the same level of abstraction, while the dependencies between these components is limited.

To quantify the number of components and their level of abstraction we define the *Component Balance* metric. This metric takes the number of components of a system and the distribution of the volume of the system over these components and outputs a score between zero and one. Interviews with experts and a case-study show that the scores of this metric correlates with scores given by experts.

The quantification of the dependencies between the components is done by a *Dependency Profile*. In such a profile, all code within a component is divided into one of four categories depending on whether a piece of code is dependent upon by, or depends on, code in other components. A large-scale experiment shows that the percentage of code in these categories is correlated with the ratio of local change within a system.

Both metrics are implemented in practice to evaluate the usefulness of these two metrics within the context of the evaluation of implemented architectures. The results of a structured observation of experts using the metrics during a period of six months and interviews with 11 experts show that there is room for improvement, but that the two metrics are considered to be useful within this context.

The combination of different research methods such as interviews, case-studies, empirical experiments and grounded theory, augmented by experiences taken from practice have lead to research results which are both valid and useful. In this thesis we lower the initial effort needed to start performing architectural evaluations by showing which concrete attributes should be taken into account, and how these attributes could be evaluated in a continuous manner. Additionally, we define and validate metrics for two of these attributes, and show that experts find these metrics useful in the evaluation of implemented architectures within practice.

Samenvatting

Softwaresystemen maken een belangrijk onderdeel uit van ons dagelijks leven. Zoals alle producten die gemaakt worden door mensen, zijn de mogelijkheden van een softwaresysteem begrensd door de keuzes die gemaakt zijn tijdens de bouw van het systeem. Deze verzameling van keuzes kan worden beschouwd als de *softwarearchitectuur* van een systeem.

Omdat de softwarearchitectuur in belangrijke mate bepaalt wat er wel en niet mogelijk is met een systeem, is het belangrijk om deze architectuur regelmatig te evalueren. Het doel van dit soort evaluaties is het vaststellen van zwakke punten binnen de architectuur, zodat men bewust een keuze kan maken deze zwakke punten te accepteren of aan te pakken.

Er zijn veel evaluatiemethoden beschikbaar die één of meer kwaliteitsaspecten van een softwarearchitectuur benadrukken. De meeste van deze methoden zijn gericht op het initiële *ontwerp* van de softwarearchitectuur. Er zijn slechts enkele methoden waarbij de focus specifiek ligt op de *implementatie*.

Als de implementatie het ontwerp precies volgt, is het geen probleem om alleen naar het ontwerp te kijken. Helaas blijkt dit in de praktijk niet altijd het geval te zijn. Sterker nog, sommige systemen worden gebouwd zonder dat er een (compleet) ontwerp aanwezig is. Ook wijst eerder onderzoek uit dat ondanks de beschikbaarheid van deze evaluatiemethoden de softwarearchitectuur van een systeem vaak niet regelmatig wordt geëvalueerd. De reden die hiervoor wordt gegeven is dat de drempel om deze evaluaties te beginnen te hoog is voor veel projecten.

Het doel van dit proefschrift is om deze drempel te verlagen door enerzijds concreet aan te geven waar men naar moet kijken tijdens een evaluatie, en anderzijds door het definiëren van metrieken die bij deze evaluatie nuttig zijn.

De algemene onderzoeksaanpak die wij hanteren kan worden bestempeld als “industry-as-a-lab”. In ons onderzoek werken wij nauw samen met de Software Improvement Group om oplossingen te bedenken en te evalueren. Om concrete oplossingen te formuleren moet een enkel kwaliteitsaspect als focus worden genomen, daarom hebben wij in dit proefschrift ervoor gekozen om ons te richten op de *onderhoudbaarheid* van een softwaresysteem.

Volgend uit het gestelde doel is het onderzoek opgedeeld in twee verschillende delen. In de eerste deel identificeren wij attributen van een geïmplementeerde softwarearchitectuur die een indicator vormen voor de onderhoudbaarheid van een systeem. Het resultaat van een analyse uitgevoerd op 40 bestaande evaluatierapporten, interviews met twee experts en een presentatie aan tien andere experts, is een overzicht van 15 architecturele attributen die experts in de praktijk als indicatoren zien voor de onderhoudbaarheid van een softwaresysteem.

Om niet alleen af te gaan op de opinie van experts hebben wij theorieën uit de cognitieve psychologie gebruikt om een bestaand model voor de complexiteit van een softwarearchitectuur uit te breiden. Met dit uitgebreide model is het mogelijk om voor elk van de 15 attributen een verklaring te geven *waarom* dit attribuut invloed heeft op de onderhoudbaarheid van een software systeem.

Met dit model en deze attributen als basis hebben wij een methodiek ontwikkeld voor een snelle diagnose van een geïmplementeerde architectuur. Deze methodiek bestaat uit 28 vragen en 28 acties verdeeld over vijf onderwerpen. Deze methodiek stelt iemand die bekend is met een systeem in staat om binnen een dag een eerste evaluatie van het systeem uit te voeren.

Het tweede deel van ons onderzoek richt zich op het ontwikkelen en valideren van metrieken gerelateerd aan twee specifieke attributen: *Balance* en *Independence*. Deze twee attributen richten zich op de twee belangrijkste bouwstenen van een geïmplementeerde architectuur, namelijk de componenten van een systeem en de relatie tussen deze componenten. Het uitgangspunt hierbij is dat een systeem in het ideale geval bestaat uit een beperkt aantal componenten die zijn gedefinieerd op hetzelfde niveau van abstractie, waarbij de afhankelijkheden tussen deze componenten beperkt is.

Om het aantal componenten en hun abstractieniveau te kwantificeren, introduceren wij de metriek *Component Balance*. Deze metriek gebruikt het aantal componenten en de verdeling van het volume van het systeem over deze componenten om tot een score tussen de nul en de één te komen. Uit de validatie van deze metriek, door middel van interviews met experts en een case-study, blijkt dat deze score gecorreleerd is met de beoordeling zoals die zou worden gegeven door een expert.

De mate van afhankelijkheid van de componenten kwantificeren wij door middel van een *Dependency Profile*. Hiervoor wordt alle code binnen een component ingedeeld in vier categorieën die duidelijk maken of code gebruik maakt van, of wordt gebruikt door, code in andere componenten. Een grootschalig experiment toont aan dat het percentage code in deze categorieën gecorreleerd is met de hoeveelheid lokale veranderingen binnen een systeem.

Om de bruikbaarheid van de bovenstaande metrieken binnen de context van het evalueren van een geïmplementeerde architectuur te testen, zijn de metrieken toegepast in de praktijk. Uit een structurele observatie van experts gedurende een periode van een half jaar en uit interviews met 11 van deze experts blijkt dat er een aantal verbeteringen mogelijk zijn, maar dat bovenstaande metrieken zeker als nuttig worden ervaren binnen deze context.

De combinatie van verschillende onderzoeksmethoden zoals interviews, casestudies, empirische experimenten en grounded theory met ervaringen in de praktijk, hebben geleid tot resultaten die toepasbaar en nuttig zijn. In dit proefschrift verlagen wij daarmee de drempel voor het evalueren van een geïmplementeerde architectuur door concreet aan te geven welke attributen belangrijk zijn en hoe deze attributen voortdurend geëvalueerd kunnen worden. Daarnaast hebben wij voor twee attributen metrieken ontwikkeld en gevalideerd, welke door experts als nuttig worden ervaren tijdens het uitvoeren van deze evaluaties in de praktijk.

Curriculum Vitae

Personal Data

Full name:	Eric Matteas Bouwers
Date of birth:	December 12th, 1982
Place of birth:	Rijswijk

Education

October 2008 - June 2013
PhD student at Delft University of Technology.

September 2005 - September 2007
Master student at Utrecht University, Master program “Software Technology”.

September 2002 - July 2005
Bachelor student at Utrecht University, Minor program “Artificial Intelligence”.

September 2002 - April 2005
Bachelor student (part-time) at Marnix Academie, Christian University for Teacher Training, Utrecht.

September 2000 - August 2002
Bachelor student at Hogeschool Domstad, Katholieke Lerarenopleiding Basisonderwijs,
Utrecht.

September 1995 - June 2000
High school (HAVO) at Christelijke Scholengemeenschap “De Goudse Waarden”, Gouda.

Work Experience

October 2007 - present
Technical Consultant at the Software Improvement Group, Amsterdam.

September 2005 - September 2007
Teaching assistant at Utrecht University, Utrecht.

APPENDIX A

Lightweight Sanity Check for Implemented Architectures
Version 1.4

A.1 Goal

LiSCIA is a lightweight sanity check for the implemented architecture of software systems. It provides a set of guidelines to perform a basic evaluation of the implemented architecture of a system. Using the questions in this document, an evaluator can critically examine the architecture that is currently implemented. Using the results of the evaluation, the evaluator can either directly perform an action to correct the implemented architecture, or justify a more in-depth architecture evaluation.

A.1.1 Pre-requisites

The input for LiSCIA is the following:

- The source code of the system to review,
- Information about the layout of the source-files on the file-system.

To get the most out of the review it is recommended to have access to the following:

- A tool to calculate the size of the source-files under review (e.g., Lines Of Code)
- A tool to calculate the dependencies between the elements of the source code under review

A.1.2 Participants

The roles in LiSCIA are the following:

1. Evaluator, the one evaluating the system

2. Expert, a person with in-depth knowledge about the system (such as a lead developer, software architect or the project leader).

The same person can fulfill both roles and multiple people can fulfill the same roles. In order to get the most out of the evaluation at least two persons should be involved in order to create discussion. Additionally, the role of the evaluator should ideally be fulfilled by someone outside of the development team of the system under review. This makes it easier to ensure an objective evaluation.

A.1.3 General overview

LiSCIA consist of two phases, a start-up phase and a review phase. The start-up phase only needs to be conducted once during the first evaluation of a system. The review phase should be conducted during every evaluation, for more details see Chapter 4.

A.2 Start-up Phase

The code of a system is divided into source files that encode some part of the functionality of the system. In order to have a good grasp of the system we need to divide the code into logical groups of functionality. Such a group of functionality is called a component.

A component can either represent some business functionality, such as Accounting and Stocks, or a more technical functionality, such as GUI and XML-processing. The evaluation can be applied to both decompositions.

Also, the evaluation can be applied to the same version of a system using different decompositions. In this way, different views on the architecture can be explored, which can lead to more insight and a better understanding of the implemented architecture.

A.2.1 Defining components

Make a list of logical groups of functionality that should be in the system. This list of functionalities should contain about 5 to 10 different core-functionalities. Usual functionalities for a typical application can be things such as “User Interface”, “Input processing”, “Administration” or “Utilities”.

Ideally, each core-functionality is a component within the system. If there are more than 15 components try to group some of the components together. For example, the components ’GUI for administrators’ and ’GUI for users’ can be grouped into a component ’GUI’.

If there is already a list of components in the documentation this list can be used.

A.2.2 Defining name-patterns

For each component, try to determine which source files belong to it by defining a pattern on the directory-/file-names of the source-files. For example, all files that implement the GUI are in a subdirectory called ’GUI’, the name-pattern for this component then becomes:

```
Component GUI, name-pattern = .*/GUI/.*
```

Note that the name-patterns for the components should be exclusive. In other words, a single file should only be matched to a single component.

A.2.3 Inventory of Technologies

Determine the technologies used within the system by:

- Listing all different file-extensions used in the system
- Mapping each file-extension to a technology

A.3 Review Phase

A.3.1 Evaluation of Source Groups

Determine whether all source-files in the system belong to a component by applying the name-patterns to the sources in the system. All source files that cannot be placed under a component fall into one of the following two categories:

- Code that can be removed because it does not implement any functionality
- Code that should be put under a, possibly new, component

When code should be put into an existing component answer the following questions:

1. Should the name-pattern be expanded or should the code be relocated on the file-system?
2. Why does the code fall outside of its desired component?

When code should be put into a new component, answer the following questions:

3. Why has this component only surfaced now?
4. Is it likely that more components will emerge?

A.3.2 Evaluation of Component Functionality

Answer the following questions about the way the sources are grouped into the components:

1. Are the name-patterns defined for the components straightforward or complicated? In other words, are the sources located in the file-system according to the components or according to a different type of decomposition?
2. Is all functionality that is needed from the system available in the components?
3. Can the functionality of each component be described in a single sentence? If not, why?
4. Do multiple components implement the same functionality? (For example, do two components parse the same messages?)
5. Does any component contain functionality that is also available in a library/framework? If so, why is this library/framework not used?

A.3.3 Evaluation of Component Size

Determine the size of each component by counting the lines of code for each file, and then summing up the lines of code of all files in a component. Answer the following questions about the size of the components:

1. Are the sizes of the components evenly distributed?
2. If not, what is the reason for this uneven distribution?
3. Is the reason in-line with the expectations about the functionality?

When previous results are available:

4. Which component has grown the most? Is this to be expected?
5. Which component has been reduced the most? Is this to be expected?
6. Is the ordering of components on size heavily changed? Is this to be expected?

A.3.4 Evaluation of Component Dependencies

Determine the dependencies between components by determining the dependencies on file-level (or lower). After this, for each dependency between two files, determine the components of the files. If no dependency between the components existed, add this dependency, otherwise add an extra weight to the dependency.

Answer the following questions about the dependencies between components:

1. Are there any circular dependencies between the components? If so, why?
2. Are there any unexpected dependencies between components? (For example, an Utilities component calling the GUI)
3. Which component depends on most other components, is this to be expected?
4. Which component is the most depended on (which component has the most incoming dependencies)? Is this to be expected?

When previous results are available:

5. Are there any new dependencies? Is this to be expected?
6. Are there any dependencies that were removed? Is this to be expected?

A.3.5 Evaluation of Technologies

Answer the following questions about the technologies:

1. Is each technology needed in the system? Can the functionality be encoded in a different technology that is used in the system?
2. Is each technology being used for the purpose it was designed for?
3. Is the combination of technologies common? Does the official documentation of the technologies mention the other technologies?
4. Is each technology still supported by an active community or vendor?
5. Are the latest versions for each technology used? If not, why?

When previous results are available:

6. Are there any new technologies added? If so, why?
7. Are there any technologies that were discarded? If so, why?

A.4 Actions and Guidelines

This section contains examples and possible actions to take for each question to ensure that in the next evaluation the question can receive a more desirable answer.

A.4.1 Grouping the sources

Q3.1.1 *Should the name-pattern be expanded or should the code be relocated?*

- Adjust the name-pattern for a component
- Relocate the source-files (by renaming or moving the directories/files)

Q3.1.2 *Why does the code fall outside of its desired component?*
No actions specified.

Q3.1.3 *Why has this component only surfaced now?*
No actions specified.

Q3.1.4 *Is it likely that more components will emerge?*

- Make inventory of new components and their place in the overall architecture
- Make a plan for the integration of these new components into the source-code

A.4.2 Evaluation of Components

Q3.2.1 *Are the name-patterns defined for the components straightforward or complicated?*

- Remodel the layout of the file-system to the functional decomposition of the system

Q3.2.2 *Is all functionality that is needed from the system available in the components?*

- Define the place where the desired functionality needs to be implemented
- See actions of 3.1.4

Q3.2.3 *Can the functionality of each component be described in a single sentence?*

- Split up a component into components with dedicated functionality

Q3.2.4 *Do multiple components implement the same functionality?*

- Merge the duplicated functionality implemented in multiple components and move it to a common component

Q3.2.5 *Does any component contains functionality that is also available in a library/framework?*

- Replace existing component by a framework/library

A.4.3 Evaluation of Component Size

Q3.3.1 *Are the sizes of the components evenly distributed?*
Q3.3.2 *If not, what is the reason for this uneven distribution?*
Q3.3.3 *Is the reason in-line with the expectations about the functionality?*

- Split components based on functionality
- Merge components based on functionality

Q3.3.4 *Which component has grown the most? Is this to be expected?*

- Limit growth by splitting up component or introducing abstraction

Q3.3.5 *Which component has been reduced the most? Is this to be expected?*

- Merge the component with an other component (if the component has become too small)
- Add more functionality (either new or old) to the component.

Q3.3.6 *Is the ordering of components on size heavily changed? Is this to be expected?*

- Re-order components by making a new inventory of the components

A.4.4 Evaluation of Component Dependencies

Q3.4.1 *Are there any circular dependencies between the components?*

- Remove circular dependency by moving dependent code into single component
- Remove circular dependency by applying inversion of control

Q3.4.2 *Are there any unexpected dependencies between components?*

- Remove unexpected call from system by offering the functionality in a different component
- Move code that is called to component that calls the code

Q3.4.3 *Which component depends on most other components?*
Q3.4.4 *Which component is the most depended on?*

- Restructure components to reflect dependency expectations
- Remove dependencies (see actions of 3.4.2)

Q3.4.5 *Are there any new dependencies?*

- If the dependencies are unexpected see 3.4.2

Q3.4.6 *Are there any dependencies that were removed?*

- If this is unexpected and the dependency is needed re-introduce the dependency

A.4.5 Evaluation of Technologies

Q3.5.1 *Is each technology needed in the system?*
Q3.5.2 *Is each technology being used for the purpose it was designed for?*

- Re-implement functionality in different technology (one that is already used more or is more suitable)

Q3.5.3 *Is the combination of technologies common?*

- Re-implement the functionality of one technology in one of the other technologies
- Re-implement the functionality of one technology in a technology that is more common to use with the other technology

Q3.5.4 *Is each technology still supported by an active community or vendor?*

- Re-implement functionality in a newer, supported technology

Q3.5.5 *Are the latest versions for each technology used?*

- Update technologies of an older version to the latest version

Q3.5.6 *Are there any new technologies added?*

- If the technologies are unexpected see actions of 3.5.1 and 3.5.2

Q3.5.7 *Are there any technologies that were discarded?*

- If this is unexpected and the technology is needed restore the technology