Operation-Centric Hierarchy-Aware Software Metrics in Component Composition Hierarchies
# Contents

## I. Introduction

1. Introduction ........................................ 3
   1.1. Research Context .................................. 4
   1.2. Research Objective .................................. 8
   1.3. Out of Scope ...................................... 9
   1.4. Outline .......................................... 9

## II. Components

2. Software Components .................................. 13
   2.1. Component Definitions .............................. 14
   2.2. Abstractions ...................................... 16
   2.3. Basic Component Modeling ........................ 18
      2.3.1. Component Notations ............................ 18
      2.3.2. Organizational Elements ........................ 19
      2.3.3. Wrapper Components ............................ 20
   2.4. Summary .......................................... 21

3. Software Component Model Categories .................. 23
   3.1. Basic Properties of Component Models ............ 24
      3.1.1. Syntax of Software Components ................. 25
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1.2.</td>
<td>Semantics of Software Components</td>
<td>25</td>
</tr>
<tr>
<td>3.1.3.</td>
<td>Software Component Composition</td>
<td>26</td>
</tr>
<tr>
<td>3.2.</td>
<td>Taxonomy</td>
<td>26</td>
</tr>
<tr>
<td>3.3.</td>
<td>Flat and Hierarchic Component Models</td>
<td>28</td>
</tr>
<tr>
<td>3.4.</td>
<td>Summary</td>
<td>31</td>
</tr>
<tr>
<td>4.</td>
<td>Frameworks and Methodologies</td>
<td>33</td>
</tr>
<tr>
<td>4.1.</td>
<td>Fractal</td>
<td>33</td>
</tr>
<tr>
<td>4.1.1.</td>
<td>Basic Level</td>
<td>34</td>
</tr>
<tr>
<td>4.1.2.</td>
<td>Configuration Level</td>
<td>35</td>
</tr>
<tr>
<td>4.1.3.</td>
<td>Conclusion</td>
<td>37</td>
</tr>
<tr>
<td>4.2.</td>
<td>PECOS</td>
<td>38</td>
</tr>
<tr>
<td>4.2.1.</td>
<td>Component Types</td>
<td>39</td>
</tr>
<tr>
<td>4.2.2.</td>
<td>Composition Language</td>
<td>40</td>
</tr>
<tr>
<td>4.2.3.</td>
<td>Conclusion</td>
<td>41</td>
</tr>
<tr>
<td>4.3.</td>
<td>SOFA</td>
<td>41</td>
</tr>
<tr>
<td>4.3.1.</td>
<td>Static View</td>
<td>41</td>
</tr>
<tr>
<td>4.3.2.</td>
<td>Behavior View</td>
<td>42</td>
</tr>
<tr>
<td>4.3.3.</td>
<td>Deployment View</td>
<td>42</td>
</tr>
<tr>
<td>4.3.4.</td>
<td>Performance View</td>
<td>43</td>
</tr>
<tr>
<td>4.3.5.</td>
<td>Conclusion</td>
<td>44</td>
</tr>
<tr>
<td>4.4.</td>
<td>KobrA</td>
<td>44</td>
</tr>
<tr>
<td>4.4.1.</td>
<td>Separation of Concerns</td>
<td>45</td>
</tr>
<tr>
<td>4.4.2.</td>
<td>Hierarchical Composition</td>
<td>49</td>
</tr>
<tr>
<td>4.4.3.</td>
<td>Component Types</td>
<td>51</td>
</tr>
<tr>
<td>4.4.4.</td>
<td>Conclusion</td>
<td>52</td>
</tr>
<tr>
<td>4.5.</td>
<td>Comparison</td>
<td>53</td>
</tr>
<tr>
<td>5.</td>
<td>Hierarchical Component Model</td>
<td>57</td>
</tr>
<tr>
<td>5.1.</td>
<td>Composition Hierarchies</td>
<td>57</td>
</tr>
<tr>
<td>5.1.1.</td>
<td>Instance Hierarchies</td>
<td>58</td>
</tr>
<tr>
<td>5.1.2.</td>
<td>Type Hierarchies</td>
<td>60</td>
</tr>
<tr>
<td>5.1.3.</td>
<td>Meta-Level Relationships</td>
<td>62</td>
</tr>
<tr>
<td>5.1.4.</td>
<td>Instance vs Type Relationships</td>
<td>64</td>
</tr>
</tbody>
</table>
### Contents

5.2. Composition Rules ................................. 66
5.3. Basic Model .................................. 71
   5.3.1. Component .......................... 71
   5.3.2. Operation ......................... 73
5.4. Conclusion .................................. 74

III. Measures .......................... 79

6. Quality Model - Related Work and Evaluation Processes 81
   6.1. Indicator Metrics .......................... 82
6.2. General Quality Models .................. 83
   6.2.1. Boehm's Quality model .............. 84
   6.2.2. McCall’s Quality model ............ 84
   6.2.3. The ISO/IEC 9126 Quality Model ....... 87
6.3. Specialized Quality Models ................. 88
   6.3.1. Bertoa and Vallecillo ............... 89
   6.3.2. Khosravi and Guéhéneuc ............. 90
6.4. Quality Evaluation Processes ............... 91
   6.4.1. Khosravi and Guéhéneuc ............. 91
   6.4.2. ISO/IEC Quality Engineering ..... 93
   6.4.3. Moraga et al. ....................... 94
6.5. Summary .................................. 96

7. Software Measurement 99
   7.1. Basic Definitions of Software Metrics ....... 100
   7.1.1. Software Measures vs Software Metrics .... 100
   7.1.2. Direct and Indirect metric .......... 101
   7.1.3. Quality Based Software Metric ......... 102
7.2. Measurement Theory .......................... 104
   7.2.1. Foundations of Measurement Theory .... 105
   7.2.2. Fundamental and Derived Measurement .... 111
   7.2.3. Extensive Measurement ............... 120
7.3. Measurement Theory based Definitions of Software Measures/Metrics 123
   7.3.1. Software Metrics .................. 124
PART

Introduction
The development of and research on component based systems has become increasingly popular in the last recent years, both in industry and academia [SH04]. The reasons for the ongoing success are many-sided. One of the reasons which contributed to the rising popularity of component based software development is certainly the steadily rising number of components accessible over the web. In [TvH00] Tras and van Hillegersberg give an insight in the conditions for the growth of a component market in general, where they point to the growth predictions of the Gartner Group [Gro98], the Giga information group [Kar97] and Ovum [Ovu98]. The latter, for instance, in 1998 estimated the size of the software component market to be $64 billion in 2002 [Ovu98].

However, the available components often differ in several properties. For example, the places where the components are provided. These reach from companies which sell their software components, so-called Commercial off the Shelf (COTS), on special marketplaces like ComponentSource.com or Flashline.com. In the other extreme software components are provided by the open source community, where these are mainly provided for free and depending on the underlying license can mostly be used without any restrictions. Components of the open source commu-
1. Introduction

Components are usually made available on special repositories such as sourceforge.net, or can be found by means of specialized search engines like merobase.com.

1.1. Research Context

But not only the components acquirable by third-partys are used within the development of component based applications. Naturally, companies have a big interest in reusing as much of their already developed software as possible in the future development of applications. This is due to the fact that reusing software artifacts can drastically reduce the development costs and time to market of applications [Voa98].

It is clear that the reusability of software artifacts does not come for free and some extra effort is needed to make a software artifact a reusable software component. In [BG90], Balda and Gustafson, introduced a cost estimation model for reuse components based on the COCOMO estimation model. By applying the reuse cost estimation model they concluded that the costs of making a software artifact a reusable software component is about 130% to 348% of the development costs of a particular software artifact.

In [MW93] the benefits of making a software artifact reusable is discussed from an economic point of view. However, they also highlighted the issues arising in industry, where software is mostly created on a project basis. As the development of reusable software components requires a high-upfront investment the economic benefit is not directly reflected in the company’s business balance sheets. This, in turn, makes it more difficult to justify the investment in them [MW93].

The reduced time to market and the reduced development costs are a basic factor for a reuse-based development of applications. Another one, however, is that these are expected to be of higher quality. Basic factors - the quality characteristics - influencing the quality of a software component are defined in the ISO/IEC 9126 [ISO91]. Beside giving a definition of the different quality characteristics such as reliability and usability, the ISO/IEC 9126 further defines metrics that can be used as indicators for a particular quality characteristics. However, the metrics defined in the ISO/IEC 9126 are neither tailored to a particular domain nor are they tailored to a
1.1. Research Context

specific model like a hierarchical composition model for component based systems. Technically speaking in the ISO/IEC 9126 it remains unclear why particular measures are defined to be indicators for a specific quality characteristic. Further, it remains unclear if these cover all the facets of a particular quality characteristic's definition.

However, the assumption that all aspects of a quality characteristic are fully covered by a particular measure is often made as well. But there is another issue that needs to be taken into account. Although a particular metric might cover a facet of a quality characteristic, it still needs to be kept in mind that this metric might have an influence on other facets as well. This, however, makes it difficult for a combination of metrics to fully cover the facets of a quality characteristic.

A commonly used way to investigate the mappings between measures/metrics and quality characteristics is to use questionnaires as a basis for a polling of specialists. Bertoa et al. [BTV06] present a method based on questionnaires to investigate on how measures/metrics indicate the usability of a software component. The method presented, however, is generally applicable and can easily be applied to assess measures for other quality characteristics as well.

So far, we did not go into further detail on what a software component is and how software components are used in the component based development. Although the term “software component” is known for several years know in the academic literature, still no clear consensus is found on what a software component is about. Currently the most cited and most accepted definition is the one introduced at the Workshop on Component-Oriented Programming (WCOP)’96 [SP97] that is attributed to Szyperski.

A central aspect of this component definition and of almost any other definition is software components of being part of a composition in order to build up more complex components out of basic ones. Heineman and Councill [HC01] further identified that a cornerstone of the component composition is an underlying component model. Mainly, all frameworks and methodologies for the development of hierarchical component based systems such as Fractal [BCL+04], Software Appliances (SOFA) [BHP06], Pervasive Component Systems (PECOS) [GCS+02] and Komponenten-basierte Anwendungsentwicklung (KobrA) [ABB+02] make use of an
1. Introduction

underlying component model. The application of these frameworks and methodologies results in hierarchical composition graphs/trees.

Anyway, the aforementioned frameworks and methodologies basically have been developed to cope a particular area. PECOS, for instance, has been developed as a component based development framework for embedded systems, more precisely for field devices. SOFA in turn addresses the facets of developing a composition architecture, to model the behavior of the component based system, to describe the quality aspects of the systems as well as it provides the ability to directly execute a SOFA based system in a specialized environment called SOFANode. Both, however, are mainly implementation/realization centric - in chapter 3 SOFTWARE COMPONENT MODEL CATEGORIES the term platform specific is used - whereas KobrA’s aim is to provide a methodology for the development of complex hierarchical component based systems. Furthermore, KobrA makes use of a view-based modeling approach where the different facets of a component are modeled in an isolated way with an underlying common system model, the so called Orthographic Software Modeling (OSM) [ASB09]. Therefore, KobrA is denoted as of being view centric and platform independent.

Other architectural frameworks like the NATO Architecture Framework (NAF) v2 provide a view based notion of the system as well. But in contrast to KobrA, where the view based notion is an integral part of the development methodology, the NAF primarily aims to provide a view based notion to highlight different facets of the system in an isolated way.

Component technologies such as Enterprise Java Beans (EJB)’s or the Component Object Model (COM) primarily use a flat composition hierarchy to compose components, where the aforementioned frameworks and methodologies are based on an hierarchical composition of components. In KobrA, for instance, the system is interpreted as the root component of the composition hierarchy. In an hierarchical refinement the system is recursively refined. In other words, the pieces of functionality provided by the root component are directly based on the pieces of functionality provided by the components in the successive level of the composition hierarchy, which in turn depend on the pieces of functionality provided by the components of the second level and so on.
1.1. Research Context

Although, the definition of software components addresses to major issues - components hiding their internal realization behind contractually specified interfaces and components being subject of composition of third parties [SP97] - the first one has been given more interest in the research community of software measurement. Basically, some software metrics have been defined to address “internal issues” of a particular component. Furthermore, software metrics of other areas are directly applied to “measure” software components. The resulting values are often directly interpreted towards a quality characteristic based on the ISO/IEC 9126 [ISO91].

The estimated level of quality, the covered pieces of functionality and the reduced cost (development, lifecycle) of a software component are driving factors of applying a particular component within someone’s own applications. In [MDL87] the authors point to a different issue of large scaled software applications. Based on their industrial experience they identified that the reduction of 60 percent of the errors in a software application does not correspond to the increased reliability of the system, as the failure rate is only reduced by 3 percent. The reasons are various. The most important however is the criticality of particular parts of the system. Usually the number of “invocations” of particular parts of the system is heterogeneous. Some parts of the system are accessed more often than others that, for instance, cover specialized and rarely used issues of an application.

Although the error reduction discussed in [MDL87] results from a general analysis of applications, the results are of major importance for the quality estimation of component based systems as well. Components in particular branches of the composition hierarchies that are central to provide the system’s pieces of functionality have a strong influence on the system’s overall quality, like the reliability. In other words, the quality of a component in the branches of major importance strongly influence the system’s overall quality. Components, instead, residing in branches not being central to provide the system’s pieces of functionality influence less the system’s functionality, as well as they have a smaller influence on the system’s overall quality.

These issues, however, have not directly been addressed in cost estimation models, such as COCOMO being based on the expected number of function points in an application [BCH*00]. In this thesis we focus on addressing the different kinds of
dependencies a component in lower levels of the composition hierarchies has on its parent-components, grandparent-components, and so on.

1.2. Research Objective

In this thesis we build the foundation for the ongoing research on specialized cost estimation models, quality measurement and improved testing capabilities for hierarchical component based systems. Based on an informal discussion of frameworks and methodologies with an underlying hierarchical component model, a common hierarchical component model is defined. We identified different kinds of relations between the components in component based systems. Components can either have instance relationships or type relationships with other components.

We further focus on the measurement theory based foundations of hierarchy-aware measures, where we introduce a suitable and commonly applicable method for a theoretical analysis of software metrics. Finally, we turn our attention on the different categories of operation-centric metrics that are inducted by the relationship types among components in hierarchical component based systems. Beside direct metrics that take the complete invocation-chain directly into account, we analyze whether recursive metrics can be defined for the particular categories. Unfortunately, not for all categories a recursive (additive) version can be defined due to the different kinds of relationships in the composition hierarchy. Thus, we introduce the parametrized measurement that allows to define recursive metrics for these categories.

Based on the categories we are able to give suitable indications of particular quality characteristics like reliability based on the delegation ratios. Delegation ratios consider the ratio of a particular category-based metric compared to the overall metric. For instance, if we take an indicator for complexity, we can determine the ratio between the complexity covered by a particular component or the composition hierarchy addressed by their underlying operations that are affected by the invocation-chain of an operation compared to the overall complexity of the complete invocation-chain. This allows us to judge whether an operation of component X covering the same functionality as an operation of Component Y for example has a higher reliability as it covers more of the provided functionality by its own.
1.3. Out of Scope

The newly introduced hierarchy-aware measurement builds the basis of our ongoing research activities on the development of a quality model and a metric suite for hierarchical component based systems. Based on the newly introduced categories, a hierarchy-aware software metric suite needs to be defined. The relations between quality characteristics and metrics need to be analyzed in detail. For instance, it is necessary to determine up to which degree the metrics can be thought of being indicators of particular quality characteristics like the ones defined in the ISO/IEC 9126 [ISO91].

A different research activity is concerned with the development of a specialized cost estimation model for hierarchical component based systems including the results gained in this thesis. A respective cost estimation model will be based on the COCOMO cost estimation model [BCH'00] and the aforementioned cost estimation models that are presented in [MW93].

These issues are not further regarded in this thesis and are differed to further research.

1.4. Outline

After presenting commonly used software component definitions in the second chapter, a software component definition tailored to the particular purpose of defining hierarchy-aware software metrics is introduced. The newly introduced software component definition is primarily based on the commonly accepted definition attributed to Szyperski and the one presented by Heinmann and Councill.

In chapter 3 **Software Component Model Categories** we turn our attention on the analysis of the different categories of component based systems. The frameworks and methodologies for the development of hierarchical component based systems being of major importance are presented in chapter 4 **Frameworks and Methodologies**. Based on the impression gained from the underlying component models of the presented frameworks and methodologies we define a commonly applicable hierarchical component model in chapter 5 **Hierarchical Component Model**.
1. Introduction

In the third part we focus on measuring facets of hierarchical component based systems. As software metrics are almost of no value unless they are interpreted towards a quality characteristics we start the second part with a brief overview of commonly used quality models and evaluation processes.

Before we give an overview of well known software metrics and metric suites in chapter 8 SOFTWARE METRICS-RELATED WORK we go into detail on the theoretical foundations of software measures and software metrics in 7 SOFTWARE MEASUREMENT. Based on the critical discussion on the theoretical foundations of software measures, a set of properties is identified that are combined to a method for the theoretically validation of software measures.

Afterwards, we discuss the different categories of operation-centric metrics in composition hierarchies. The three categories - local, shallow and deep - identified in chapter 9 HIERARCHY-WARE MEASUREMENT are discussed in detail, where amongst other issues the theoretical properties of the operation-centric metrics of each category are introduced. The category-based considerations are set in relation to each other resulting in a delegation-ratio based indication of quality characteristics such as reliability.

Unfortunately, it is not possible to define additive metrics for some of the subcategories of the deep operation-centric category due to the different kinds of association types among the components of the composition hierarchies. Therefore, we introduce the parametrized measurement that allows an additive consideration of the respective categories in chapter 10 PARAMETRIZED MEASUREMENT.

We close this thesis with chapter 11 EPILOGUE, where we give a summary of this thesis and a brief overview of the current and future research activities in this research area.
PART II

Components
Chapter 2

Software Components

In historical terms “software components” have firstly been discussed at the famous NATO Software Engineering Conference at Garmisch-Partenkirchen (Germany) in 1968. In his keynote Douglas McIlroy presented his early vision of using software components to cope with the arising problems of software development. “Families of routines for any given job” should be provided by the software component industry, from which a purchaser could choose the component “tailored to his exact needs” [McI68].

For more than 40 years have the development of software components and associated technologies been in the focus of research and industry. Several component technologies, methodologies and frameworks for the development of components and component based systems have been elaborated since these early days. However, it took till 1996 to find a definition of software components that tends to be commonly accepted in the research community. Further important definitions of software components are found in the literature. At the top of this chapter we aim to give an overview of the most important definitions of software components and issues related to these.

In recent years the Unified Modeling Language (UML) has become the de-facto standard for graphically modeling modern software systems. The UML provides
2. Software Components

several diagram types to model the facets of software systems. One of the diagram types - the component diagram - addresses some of the issues related to the structural relationships of software components. In section 2.3 BASIC COMPONENT MODELING an overview on how software components are graphically modeled with UML component diagrams is given.

2.1. Component Definitions

Since McIlroy introduced his vision of modern software development based on software components many researchers presented their definitions on what they think the term “software component” is about. Unfortunately, almost every researcher has his own idea of what a software component is.

A definition of the term that tends to be commonly accepted by the research community is the one attributed to Clemens Szyperski. This definition, however, has been elaborated at the WCOP co-located at the European Conference on Object-Oriented Programming (ECOOP) in 1996 and defines a component as [SP97]:

“A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to third-party composition.”

Currently, the definition attributed to Szyperski is the one mostly being cited in recent research. However, other definitions being of importance are the ones from Booch [Boo87] presented in 1987 or the definition given by the Object Management Group (OMG) [Obj03] in the UML specification (version 1.5).

Booch defines a software component to be a

“[… ] logically cohesive, loosely coupled module that denotes a single abstraction [Boo87]”

The OMG, instead, defines a software component to represent

“a modular, deployable, and replaceable part of a system that encapsulates implementation and exposes a set of interfaces [Obj03].”

Booch’s definition of a software component is the one being most abstract. It mainly points to the issues that components need to be a logically cohesive and loosely cou-
2.1. Component Definitions

pled module. In McIlroy's early vision of software components and related technologies he identified potential areas of applications where software components can be extracted from already existing implementations. Numerical approximation routines have been one of the most promising candidates. In terms of Booch's definition, abstractions of such routines are logically cohesive as they represent some restricted functionality. They are further loosely coupled as they cover the restricted pieces of functionality by their own.

By analyzing whether an already existing implementation of a particular functionality is logically cohesive and loosely coupled, we can identify whether the considered functionality is a candidate of being extracted as a software component. However, some basic questions remain uncleared. How can both issues be measured? Having identified metrics indicating the degree of logical cohesion and loose coupling, up to which degree of both metrics is it worth to extract a piece of functionality as software component?

Although, fulfilling Booch's definition of a software component, a particular software unit does not necessarily need to be easily deployable, as it is claimed by the definition of a software component in the UML 1.5 specification. A central issue that is not addressed by Booch's definition is how to describe the relationships to other parts of a system or components. Both, the OMG's and Szyperski's definition of a software component address this issue.

Szyperski explicitly claims the necessity of interfaces specifying the contract between the component and its environment. The OMG's definition is less specific on that point as it only claims the existence of "exposed interfaces". Software components of being deployable is a further similarity between both definitions. However, a central issue that is exclusively addressed by Szyperski, is components of being units of composition and subject of third-party composition.

Although, composition is a central aspect of Szyperski's definition, it remains unclear on how composition can be performed. A definition of software components taking this issue into account is given by [HC01]. The authors, Heineman and Councill, define a component as being a
2. Software Components

“[…] software element that conforms to a component model and can be independently deployed and composed without modification according to a composition standard.”[HC01]

In the latter case, Heineman and Councill emphasize quite specifically the fact that a composition of components needs to be based on a composition standard. Since a composition standard, in turn, is at least influenced by the underlying component model, this definition goes far beyond the one attributed to Szyperski. Heineman and Councill’s definition does not only define what a component is, but explains how these can be constructed and how these can be composed or assembled [LW05a].

Based on the definitions of software components presented so far, we give a definition of software components that addresses best our purposes in this thesis. These are presented throughout the rest of this chapter. The newly introduced definition of a software component is mainly based on the definitions of Szyperski and Heineman and Councill. However, interfaces that are central to the definition of Szyperski and the OMG are interpreted as purely organizational elements (see section 2.3.2 Organizational Elements). Basically, operations are defined as underlying elements of interfaces that are responsible for the dependencies between components. Therefore, an operation centric rather than interface centric definition for software components is used:

**Definition (Software Component).** A software component is a unit of composition defining a contract with its context. The contract between the software component and its context defines the pieces of functionality provided by the component (provided operations) as well as it defines the pieces of functionality that need to be provided to the component to fulfill its obligations (required operations). The dependencies between the component and its context are fully described by its contract. A software component needs to conform to a component model and can be independently deployed and composed without modification according to a composition standard.

2.2. Abstractions

Basically, the aim of developing software components out of already existing implementations, as proposed by McIlroy [McI68], is to reuse the implementations in
2.2. Abstractions

other systems as the one they have been developed for. Generally, it is not necessary to know anything about the implementation details of a software component to reuse it. The definitions of software components presented in the previous section refer to this issue.

Mainly three abstractions - black box, white box and gray box abstraction - are found in the literature that refer to the visibility of the implementation behind an interface or specification. Generally, in the black box abstraction the information being visible from “outside”, without relying on the implementation details, are described. In terms of software components, the black box abstraction describes what pieces of functionality are provided and what pieces of functionality are required by the component to fulfill its contractually specified obligations. A software component’s black box description is also called specification of the component.

In contrast, by referring to a software entity as a white box abstraction, its implementation details are taken into account. This means, that we rely on how a software entity realizes its specified obligations. As a white box abstraction provides information on how the provided pieces of functionality are realized, alternatively it is called its realization.

Gray box abstractions are somewhere in between, as they reveal a controlled part of their implementation [Szy02, p.41]. The gray box abstraction plays an important role in the analysis of hierarchical component based systems.

Composition is a central aspect of the component’s definition. Software components are composed to larger entities covering a broader range of functionality. The recursive composition of software components leads to a tree or graph based representation. Usually a heterogeneity of software component abstractions is found in component based systems. Beside components for which both specification and realization are available, others only provide access to their black box abstraction. In figure 2.1 schematically a resulting graph based structure is given.

The component at the top of the graph - CompA - is internally composed of several components some of which are available as white box abstractions, others only as black box abstraction. More in detail, we assume that CompA exposes its local realization. From a composition point of view only partial information on how its provided pieces of functionality are realized is available. Therefore, CompA needs to
2. Software Components

be thought of being a gray box abstraction. In contrast, CompD is fully composed of components exposing their realization. Therefore, by further assuming that CompD exposes its local realization, from a composition point of view CompD can be interpreted as being a white box abstraction.

2.3. Basic Component Modeling

After having identified a basic definition of software components and different kinds of abstractions in the previous sections, we give an overview on how software components are currently modeled by the use of a graphical modeling language. Modern software systems are modeled by use of the UML which can be regarded as the de-facto standard in graphical modeling of software applications [CD00].

2.3.1. Component Notations

The UML uses different notations to describe the structural contract defined by a component. In the component box notation [Obj09a, p. 150] compartment boxes are used to define the required and provided interfaces. Addressing the OMG’s definition of software components [Obj03], interfaces are used to describe the contract between the software and its context. This particular notation has its focus on presenting the component’s dependencies to required and provided interfaces. Further,
an additional realization compartment box is added, if the component’s realization is regarded as well.

The explicit notation is a second notation to model a component, which has its focus on graphically describing the relationship between the component and its interfaces. The use of the latter allows a much easier visual distinction between the different kinds of interfaces used and their comprised operations. In figure 2.2 an explicit representation of a component - Order - taken from [Obj09a, p. 150] is given.

Further, the UML’s nested representation of a component (see Figure 2.3) allows to describe the internal details - the realization - of a component. Beside the black box details, the nested representation allows to describe the delegation of interfaces and the dependencies to other components or classes.

Based on the underlying definition in [Obj03], the UML interprets interfaces as fundamental elements of a component. However, the UML provides a second element - ports - which aim to hide the internal implementation details from the
2. Software Components

component’s environment. In our definition of software components we avoid the use of interfaces, ports and other organizational elements which aim to hide the implementation details from the component’s environment. As our aim is to consider the properties of operation-centric hierarchy-aware software metrics in component compositions, organizational elements are of almost no interest and are not further regarded.

One of the reasons of not further regarding organizational elements such as ports and interfaces is that the component’s interfaces are usually defined by the developer/modeler, where he groups provided and required operations he thinks of belonging together. A simple workaround for using operations grouped in a different way than the interface defined by the requester, is to use wrapper components.

2.3.3. Wrapper Components

In McIlroy’s vision of software components he was thinking of obtaining software components from catalogs. Since the presentation of his vision many technologies like the World Wide Web (WWW) evolved. Goods, such as books and music can be obtained from particular virtual marketplaces. For an easier identification of particular goods being of interest for a user, search engines are used. They match a particular request with the goods provided on the virtual marketplace and try to identify the “optimal result”. Specialized marketplaces and search engines, such as Koders, Krugle and Merobase allow to find software entities, fitting best for a given request. Merobase, for instance, provides the ability to search for software components by e.g. defining a particular interface with the operations that need to be provided by a candidate component.

Although particular candidate components provide the requested pieces of functionality, their interfaces and those defined in the request do not necessarily need to correspond to each other. If a requested component, however, should be integrated in a component composition, the interfaces would need to correspond to each other. As a practical solution a wrapper component could be used. This wrapper component enables the elements of the composition to invoke the operations of the candidate component. A graphical representation of this issue is given in figure 2.4.
2.4. Summary

The dependencies between a component in the composition hierarchy and a component that is going to be integrated in the hierarchy are not effected by the use of a wrapper. From an abstract point of view, wrapper components only forward the invocation of a particular operation to the respective component. The responsibility covered by a wrapper component does not add additional functionality to the composition hierarchy. Therefore, we decided to define the contract between the component and its context through its required and provided operations rather than interfaces, as it is done in Szyperski’s definition. The reason to modify Szyperski’s definition towards an operation oriented definition of the contract, rather than an interface oriented definition, is our aim to highlight, analyze and measure the dependencies in composition hierarchies of view centric platform independent component based systems.

2.4. Summary

At the top of this chapter we presented the most important definitions of the term “software component”. Based on the definitions of Szyperski [SP97] and Heineman and Councill [HC01] we gave a combined operation-centric definition of the term “software component”.

In Szyperski’s definition the contract between the software component and its environment was specified through provided and required interfaces. In this thesis we aim to consider the properties of operation-centric hierarchy-aware software metrics in component composition hierarchies. Therefore, we modified the original definition by requiring that a component needs to specify its required and provided
2. Software Components

operations rather than its provided and required interfaces. To analyze the dependencies between the component and other elements considering operations rather than interfaces is sufficient, as in practice interfaces are mainly used as purely organizational elements. These do not add any additional functionality and are used to specify how software elements are connected from an organizational point of view with each other. The functionality provided by a component and any other element, however, is established through the realizations of the corresponding operations.

Further, we presented different kinds of abstractions of software components that define the information details available of a particular software component and other software units. Basically, we have identified that a single locally regarded software component has two levels of abstraction - black box and white box - where three levels of abstractions are known if a component is regarded from a composition hierarchy point of view - black box, gray box and white box.

However, the introduced software component definition basically describes what a component is. The details on how a component composition is performed and what additional information, e.g. the component's realization, need to be taken into account is not further addressed by the given definition. The second part of the introduced definition requires a component to be based on a component model. An overview of hierarchical component models and their encapsulating frameworks is given in chapter 4 Frameworks and Methodologies. The categorization of component models is discussed more in detail in the next chapter.
In the previous chapter we introduced an operation-centric definition for software components that is based on the definitions of Szyperski [SP97] and Heineman and Councill [HC01]. The second part of our component definition addresses the issue that a component needs to be based on an underlying component model.

In [LW05a] the semantic relationships of components have been identified to be a basic property of a component model. Further, the definition of a particular syntax as well as the definition of the composition of components have been identified as basic properties. However, in their analysis of frameworks, methodologies and Architecture Description Language (ADL)'s for component based systems, beside these basic properties, other properties were identified as being of importance. Some of these additional properties are used in conjunction with the basic properties to categorize component models in a taxonomy [LW05b].

Beginning with the basic properties of component models we further analyze their underlying composition structure and identify two major categories. Component models can generally be classified as having either a flat or a hierarchic composition structure [FA08]. In short, hierarchic component models can be interpreted
3. Software Component Model Categories

as allowing a recursive composition of components yielding complex components with multiple composition levels. In contrast, flat component models restrict the component composition to making use of a single level only.

In the following section an overview of the basic properties of component based systems is presented. The identified properties build the foundation for the categorization of component based systems resulting in a respective taxonomy that is presented in section 3.2 TAXONOMY. In section 3.3 FLAT AND HIERARCHIC COMPONENT MODELS component based systems are further categorized in flat and hierarchical component models resulting in a refined taxonomy that is presented at the end of the respective section.

3.1. Basic Properties of Component Models

In the literature and in practice the term “component model” is used with different meanings. For instance, the term component model in the COM is used to describe a methodology and a model to describe and enable the interprocess communication between components. Based on an interface standard, the COM allows language-neutral objects to be defined that can be used in environments different from the one they have been created for. For instance, the COM as it is used in Microsoft’s Windows-family of Operating Systems (OS) enables COM-based software components to communicate [Mic08]. COM-components are directly related to the system (interpreted as the root of the composition hierarchy) resulting in a flat composition hierarchy.

Fractal (see eponymous section 4.1) basically aims to provide a methodology to develop runtime reconfigurable systems. In order to achieve this goal, use of a categorization of component types is made, each of them covering a different type of basic controlling-functionality. The strict assignment of responsibilities to particular component types and the restrictions involved due to this assignment are thought of being an integral part of the Fractal component model.

Another example is the KobrA component model (see section 4.4 KOBRA), that provides the ability to describe several dimensions (“separation of concerns”) of a component. This allows the component and its interaction with other components in an
3.1. Basic Properties of Component Models

architecture to be fully described by defining an operational, a structural and a behavioral view for its black box and white box parts. However, as well as a component model, KobrA provides a full methodology for the development of a product-line oriented component-based system.

Based on a comparison of the most widely used component models in [LW05a] Lau and Wang identified three criteria that a component model should consist of. A component model should define

- **the syntax of the components**, i.e. how components are constructed and represented,
- **the semantics of components**, i.e. what components are meant to be and
- **the composition of components**, representing how components are composed and assembled.

Ideally, the composition of components needs to be based on a specialized composition language such as the one presented by [LAN00]. However, they realized that the underlying component models of common frameworks and methodologies like JavaBeans, EJB, COM .NET, and CCM do not make use of any composition language.

### 3.1.1. Syntax of Software Components

Generally, the syntactic rules for a component should be defined by a component model. Obviously, this is the language used for constructing components [LW05a]. For most of the component models a formal language similar to a programming language tends to be used. However, KobrA constitutes an exception as it makes use of UML diagram types to describe components and the interaction amongst them.

### 3.1.2. Semantics of Software Components

The semantics of a component model needs to define what a component is meant to be and of what parts it should consist of. Generally, a software component is meant to be a *software unit* consisting of a *name* and an *interface* describing the
3. Software Component Model Categories

services/ operations provided by the component [LW05a]. These can be realized by making use of services/ operations provided by other software units. This means that in order to describe a component, information about its provided and required functionalities/services must be provided.

3.1.3. Software Component Composition

Software component compositions describe how components can be composed into larger entities which may or may not be regarded as components themselves. [SP97]. Ideally, a specialized composition language is provided for this purpose. While some component models make use of a specialized ADL-like description language, others make use of a modeling language like the UML to describe the composition of components. However, in [LW05a] component frameworks like EJB, COM and JavaBeans are characterized as not making use of any composition language.

In this point, we partially disagree with the authors of [LW05a]. COM based components, for instance, are directly connected to a “system”. The integration of components in such systems is only possible by making use of predefined specifications of the systems' interfaces, like Microsoft's COM. Another example are EJB's that are directly connected to a “component container”. Interpreting the system and the component container as an “ultimate high-level” component in a composition graph, the connection between the component and system/container can be interpreted as some kind of flat composition (see also section 3.3 Flat and Hierachic Component Models).

3.2. Taxonomy

In addition to the basic properties that all component models have in common, the authors of [LW05a] identified a set of properties that are only provided by a small number of component models. Based on these, a taxonomy of component models is defined and is presented in the following.

One of the most important properties for categorizing component models is the point in time when components can be composed to larger entities. Generally, com-
ponent models allow component composition either in the design or the deployment phase. Several additional properties are found for each of the composition stages.

Composition in the design phase includes the construction and allocation of components to particular repositories, if available. Some component models make use of component repositories, from which newly added components can be retrieved for the further composition into more “complex” components. Other component models have been identified by Lau and Wang that do not make any use of repositories at all or if repositories are used they do not allow the newly added components to be used further.

In contrast to the design phase where components only represent templates and therefore can be interpreted as being “stateless”, instances of components are used for composition in the deployment phase. Such component instances are both, executable and stateful.

A structured overview of the properties used by Lau and Wang [LW05b] to categorize component models is given in the following:

- **DR** Components can be created during the design phase and are deposited in a repository.
- **RR** Components can be created during the design phase and are retrievable from a repository.
- **CS** Composition takes place during the design phase.
- **DC** Components that have been designed during design phase can be deposited in the repository.
- **CP** Composition takes place during the deployment phase.

Based on these properties several combinations would be thinkable. However, only four practical kinds of combination have been identified leading to four categories of component models [LW07]:

The component models of the first category - *design without repository* - have the characteristics DR and CS, which means that new components can be created during the design phase and a composition of components is possible during the design phase, as well. Components, however, can not be retrieved from the repository nor it is possible to deposit newly designed composed components in the repository.
3. Software Component Model Categories

The second category - design with deposit-only repository - combines the properties DR and CP. In other words, components can be created and deposited in a component repository but newly created components are not retrievable from the repository nor can “complex” components be built in the design phase. Composition of “complex” components is confined to the deployment phase.

Except that components are not composable during deployment phase, the component models of the third category - design with repository - fulfil all of the other characteristics. New components can be created and deposited in a repository and can be retrieved from the repository in order to build more complex composed components during the design phase. The composed components can then be deposited in a repository as well.

The last category of component models - deployment with repository - is based on the deployment phase composition of components. Component models of this category allow the newly created components to be deposited in a repository as well.

In the following table an overview of the characteristics fulfilled by the presented categories is given. The assignment of the most important component models to one of the categories is given in the second column of table 3.1.

<table>
<thead>
<tr>
<th>Category</th>
<th>Component Models</th>
<th>Characteristics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Design without repository</td>
<td>Acme-like ADLs, UML2.0, PECOS, Fractal</td>
<td>DR ✓, RR ×, CS ✓, DC ×, CP ×</td>
</tr>
<tr>
<td>Design with deposit-only repository</td>
<td>EJB, COM, .NET, CCM, Web Services</td>
<td>DR ✓, RR ×, CS ×, DC ×, CP ✓</td>
</tr>
<tr>
<td>Design with repository</td>
<td>Koala, SOFA, KobrA</td>
<td>DR ✓, RR ✓, CS ✓, DC ×, CP ×</td>
</tr>
<tr>
<td>Deployment with repository</td>
<td>Java Beans</td>
<td>DR ×, RR ×, CS ✓, DC ×, CP ×</td>
</tr>
</tbody>
</table>

Table 3.1.: A taxonomy based composition [LW07]

3.3. Flat and Hierachic Component Models

As presented in the previous section, the composition of components can either be accomplished during design or deployment time. Additionally, further properties
3.3. Flat and Hierarchic Component Models

like depositing and reusing the newly created components in and from a repository need to be taken into account. These are used as supplementary distinction points to build a taxonomy for component models.

Although, some basic properties of the component composition are taken into account in Lau and Wang's taxonomy, many other properties exist that might be usable for a categorization, as well. For example, in [FA08] we focused on the resulting composition structure as the basis for a taxonomy of component models, where we recognized component models to have either have a \textit{flat} or a \textit{hierarchic} composition structure. In [BHP06] a similar distinction is recognized. [BHP06], however, uses a different terminology which distinguishes between \textit{primitive} and \textit{composite} component models.

\textbf{Definition} (Flat component models). \textit{In a flat component model components are regarded as indivisible (black box) entities which can only be composed at one level to create systems.}

\textbf{Definition} (Hierarchic component models). \textit{Hierarchic component models allow components to be composed of other components in a recursive manner, yielding architectures with an arbitrary number of composition levels.}

Hierarchic component models can be further refined and a categorization into two distinct sub-groups can be defined. The first group is composed of component models that can mainly be regarded as being \textit{platform specific}. The respective components need to be realized in a particular manner or need to be based on a structure given by the underlying component model, for example by implementing a particular interface. Independently of how the components need to be defined, in respect of the underlying component model, the “source code representation” of the components and the composition graph is the central view for defining these.

In contrast, the second group is comprised of those component models which can mainly be regarded as being \textit{platform independent}. Components and composition graphs are developed by making use of a particular modeling language like the UML.

Some component models, such as Fractal [BCS04], are mainly platform specific, but enhancements to components and composition graphs to be modeled exist as well [AAB+07]. Although enhancements of platform specific component models to
3. Software Component Model Categories

A more model based approach exist, the nature of these component models, of being platform specific, is unchanged.

The platform independent component models, however, can be further refined. A sub-category of component models gaining major interest in the research community are view based component models. Although view based frameworks, such as the NAF exist and are gaining major interest in industry, KobrA, to our knowledge is currently the only framework or methodology being both platform independent and view centric.

In contrast to SOFA that makes use of different views to express particular facets of an application from different point of views without giving advice how to develop the underlying content of the respective views, the view centric consideration is an integral part of KobrA’s enclosing methodology. Thus it is categorized as the only framework or methodology for the development of view centric platform independent component based systems.

The refined taxonomy taking the categorization of Lau and Wang as well as the categorization frameworks and methodologies with an underlying flat or hierarchical component model into account is presented in table 3.2. Frameworks and methodologies with an underlying hierarchical component model are further categorized in being either platform specific or platform independent. The latter are additionally categorized in either being view centric or not.

<table>
<thead>
<tr>
<th>Component model</th>
<th>Flat</th>
<th>Hierarchic</th>
<th>View centric</th>
<th>Taxonomy Category</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Flat Platform Specific</td>
<td>Hierarchic Platform Independent</td>
<td>View centric</td>
<td></td>
</tr>
<tr>
<td>EJB</td>
<td>✓</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>COM</td>
<td>✓</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>CCM</td>
<td>✓</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>PECOS</td>
<td>×</td>
<td>✓</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>Fractal</td>
<td>×</td>
<td>✓</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>Koala</td>
<td>×</td>
<td>✓</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>SOFA</td>
<td>×</td>
<td>✓</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>KobrA</td>
<td>×</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Table 3.2.: Assignment of component models to composition categories
3.4. Summary

Several component based frameworks and methodologies are found in the literature. In this chapter we introduced a characterization of these frameworks and methodologies based on the results of Lau and Wang. They basically classified component models as of being “deployment with repository”, “design with deposit-only repository”, “design without repository” and “design with repository”. Based on this categorization Lau and Wang analyzed the most important frameworks and methodologies with an underlying component model resulting in a respective taxonomy.

The results of Lau and Wang have been used as starting point for a further categorization of component models and their encapsulating frameworks and methodologies according to the classification of flat and hierarchic component models. The frameworks and methodologies with an underlying hierarchic component model are further categorized in platform specific and platform independent component based systems, where KobrA additionally has been identified to be view centric.

Before we turn our attention on the definition of a common component model for hierarchic component based systems, we give a brief overview of the most important frameworks and methodologies with an underlying hierarchical platform specific component model.

Frameworks and methodologies with an underlying flat component model such as COM, .NET or CCM are not further regarded in this thesis. The composition model of these component models, however, can be thought of being a specialization of the hierarchical component models.
The characterization of frameworks and methodologies with an underlying hierarchical component model has lead to the categories of platform specific and platform independent component models. In this chapter we give a brief overview of frameworks and methodologies with an underlying platform specific component model. The component models that are presented in this chapter are used as foundation for the development of a common component model that is introduced in the next chapter.

4.1. Fractal

The Fractal component model [BCS04] is one of the most important platform specific hierarchic component models. Fractal is mainly intended to implement, deploy and manage complex software systems such as embedded or middleware systems [BCL+04]. In contrast to KobrA's component model (see section 4.4 KobrA) that uses a view-centric notion to identify and describe the different facets of a component, the Fractal component model defines only a single viewpoint for the development of components and/or complex architectures. Admittedly, the objective target
4. Frameworks and Methodologies

of Fractal is different from the one of KobrA as it mainly aims to provide an efficient component model for the development of dynamically reconfigurable components/architectures.

The aforementioned goals of Fractal, however, motivated the following main features [BCL+06]:

**Composite components:** In order to provide a uniform view on the development of complex components/applications at various abstraction levels the Fractal component model uses composite components.

**Shared components:** This special kind of component is used to model resources whose encapsulation is mentioned throughout the development process.

**Introspection capabilities:** As the runtime management of complex architectures is one of the primary goals of Fractal, it focuses on monitoring running systems by providing components with introspection capabilities.

**Configuration and reconfiguration capabilities:** Beside the introspection capabilities, mainly configuration and reconfiguration capabilities are provided in order to deploy and dynamically reconfigure a running application.

In [BCS04] the Fractal component model is presented as an open component model that provides an “extensible system of relations between well defined concepts and corresponding APIs that Fractal components may or may not implement”. The extensible specifications can be ordered by an increasing level of control capabilities.

### 4.1.1. Basic Level

Components at the lowest level of control capability do not provide any introspection or intercession functions to other Fractal components. Access to their provided functionality is only possible by invoking the respective operations on its interfaces. The components interfaces are therefore used as access points to a component that implements a particular language interface.

Generally, the Fractal component model is not specific about how the basic ideas are implemented. However, an interface driven approach is the only one currently found in the literature. Therefore, the Fractal specification introduces three interfaces (Name, NamingContext and Binder) in order to discover components and their
execuable operations. Basically, components are registered at a particular naming context of another component, for instance in the interface driven approach through operations provided in the NamingContext interface, where a name is generated and assigned to the registered component. Components are discoverable through the naming context as well. In the presented approach, references to a discovered component are returned. The discovered component can be bound through the Binding interface and its operations can directly be invoked.

4.1.1.1. Introspection Level

At this level of control capability, Fractal components provide introspection functions that allow the external features of the component (i.e. its boundaries) to be discovered (i.e. introspection). Components generally can be regarded as black boxes, hiding their internal implementation behind an externally visible and accessible interface (the general form of an interface is meant here) or as white box component. White box components provide a view on their internal realization. In Fractal a distinction between both views is made. Further, Fractal distinguishes between interfaces in being either required (“server interfaces”) or provided interfaces (“client interfaces”). This kind of distinction is also found in other component models like KobrA, but also more generally in modeling languages like the UML.

Fractal components at this level of control capability need to provide two different kind of introspection functions. First, related to the interface driven approach an introspection function to get the components interfaces and second, one to introspect the interfaces themselves need to be provided. For details on how this is realized in the interface driven approach we refer to [BCS04].

4.1.2. Configuration Level

In contrast to components with basic introspection capabilities, components at the configuration level of control capability provide operations to introspect and reconfigure the internal capabilities of a component that are provided by sub-components. Generally, a “complex” Fractal component, can be thought of being an envelope (in [BCS04] the terms membrane and controller are used) providing access to its internal realization, which in [BCS04] is called the content of the component.
4. Frameworks and Methodologies

4.1.2.1. Component Types

With the composition of sub-components, complex functionality can be realized and is invocable through operations of the enclosing component. The enclosing component, however, is responsible for controlling the directly enclosed sub-components and can start, stop, configure and reconfigure the sub-components in the composition hierarchy of its content. However, the Fractal specification [BCS04] for this level of control capabilities distinguishes between different kind of components

**composite components** exposing all of their content,

**primitive components** that do not expose their content but provide at least one control capability, for instance through respective interfaces and

**base components** that do not provide any control capabilities.

4.1.2.2. Controller

In order to obtain control of a component, control capabilities addressing the basic properties of the component need to be provided. The Fractal specification [BCS04] distinguishes four kind of control capabilities:

1. In order to get access to a component’s state that is generally thought to be stored in particular attributes, control capabilities on the component’s attributes are required. In the interface driven approach an interface *Attribute-Controller* is introduced which provides getter and setter methods that can be invoked to manipulate the component’s state.

2. Control on the binding of the component’s externally visible interfaces, its provided/client interfaces, is gained through a respective controller.

3. Control on the sub-component’s, instead, is gained through a content controller that allows sub-components in the realization of the controlling component to be added and removed.

4. Finally, a life cycle control capability is needed to gain access to the sub-components life cycle as these might need to be started or stopped. Control on starting and stopping components is at least necessary when the sub-components are executed as separate thread or process.
4.1. Fractal

4.1.2.3. Internal Component Structure / Composition

At this level of control capabilities a Fractal component can be characterized as having a controller part that is realized by the so called membrane of the component and the component’s content. In a white box view, an overview of how the provided pieces of functionality of the component are realized by use of its encapsulated sub-components is given (see Figure 4.1).

![Diagram of Internal View of a Fractal component](BCS04)

Figure 4.1.: Internal View of a Fractal component [BCS04]

As it can be seen in figure 4.1 the components interfaces are distinguished as being either internal or external. The external interfaces provide access to the functionalities of the component and can only be accessed from outside the component. In contrast, the internal components are only accessible by the component’s sub-components.

4.1.3. Conclusion

Fractal is a widely used and very powerful component development framework which mainly aims to provide capabilities for the runtime configuration and re-configuration of an application. Components are given particular control capabilities in order to give full access to the internal realization of the sub-components. Depending on the application, particular control capabilities are required in order
4. Frameworks and Methodologies

to support a safe reconfiguration. For instance, life cycle control capabilities are needed if the sub-components are executed in different threads or processes.

The reconfiguration strongly influences the reliability of an application. The use of Fractal’s capabilities concerning the reliability of an application is further analyzed in [LLC07]. Constraints on the reconfiguration are also introduced in [LLC07], for instance the integrity constraint, that allows a reconfiguration only if the resulting state of the application is still consistent.

Although Fractal’s aim is to provide a framework for reconfiguration, a porting to a platform independent view based on the UML is found in [AAB+07]. For describing Fractal components with the UML several views are used, which are similarly to the ones used by KobrA. Fractal’s enhancement to a view based platform independent model is closely related to the one defined in KobrA. However, a methodology on how the different facets of the system/component need to be modeled is not provided.

Generally, Fractal specifies which properties components with different control capabilities should provide. But, Fractal is not specific about how this should be realized, although in Fractal’s specification an interface based approach is used where component’s having a particular control capability should be inherited from a particular interface. Due to the similarity between both underlying component models, KobrA can be applied to describe the composition hierarchy of Fractal components. Further, KobrA can be used as basic methodology for the development of Fractal conform applications.

4.2. PECOS

Fractal is mainly concerned with the reconfiguration of applications during runtime, PECOS, instead, has its focus on the development of applications for a special kind of embedded systems known as “field devices” [WDN05]. Several challenges are faced in the development of embedded software applications (see [GCS+02, pages 11–12]) and PECOS is mainly concerned with the provision of a component model that helps to reduce the shortcomings of scheduling problems on field devices.
4.2. PECOS

Similar to the KobrA component model, PECOS is a hierarchical component model with a “top-level” component representing the modeled/designated application. In the special case of PECOS, the field device is thought to be the “ultimate top-level” component encapsulating several “top-level” components which represent the applications executed on a field device.

4.2.1. Component Types

Like in all other hierarchical component models a distinction between component types is found for PECOS. Component types are distinguished along two dimensions resulting in an encapsulation centric and a control centric dimension. The former, however, is generally valid and is also found in all of the analyzed hierarchical component models. Along this dimension components are distinguished as being either composite components which describe components that encapsulate other components or leaf components describing components that do not contain any other components [NAD+02].

Based on [NAD+02] component types along the control dimension are classified as:

**Passive Components** Passive components are regarded as not having their own thread of control. As a consequence this special kind of component is thought of having a restricted piece of behavior that executes synchronously and completes in a short time-cycle.

**Active Components** In contrast to passive components, active components are defined as having their own thread of control. Therefore, components of this kind are executed in parallel to the encapsulating component. In case of applications being developed for field devices, this kind of component is mainly used to model either very fast or very slow activities.

**Event Components** Like active components, event components have their own thread of control, but in contrast to active components these are triggered by an external event, for instance by certain pieces of hardware like sensors.

A PECOS component is completely described by its name, a set of “property bundles”, its ports and a description of its behavior. Properties stored in “property bundles” aim to describe non-functional properties of a component such as the worst
case execution time. Ports, in contrast, represent a description for the relations to other components. In detail, ports, are described through their particular name, the type of data, respectively its value ranges and finally the direction of the port (in/out, or in other words required/provided) [GCS’02].

The ports of a component represent the endpoints of a relation between components. The relation itself, however, is represented through a so called connector. Connectors in PECOS are thought of representing a data-sharing relationship between ports. Like Fractal, the PECOS component model allows the component’s behavior to be described. But in contrast to Fractal components that consist of a controller part - the so called membrane (see section 4.1) - the PECOS component model uses sub-typing of particular classes that provide functionalities to initialize, execute and synchronize a particular behavior.

4.2.2. Composition Language

In contrast to the other analyzed component models, the PECOS component model defines a formal composition language called Component Composition Language (CoCo) [GCS’02] which is an implementation of the previously described properties. It is used to define components and the connection between these. Beside the formal definition of components and their interaction, the definition of non-functional properties of the component are of main interest in the component model in general and the CoCo definition language in particular. Non-functional properties that are described, are for instance the definition of scheduling the component’s sub-components and the component’s memory consumption.

Fractal’s motivation for the development of a component model is mainly related to the reconfiguration of running applications/systems. The estimation of the degree of reliability of a reconfigured application is, therefore, of primary interest [LLC07]. In contrast, PECOS’ aimed to provide a component model mainly concerned with the development of applications for field devices. A major problem for applications on such a device are the control of scheduling the applications running on this device and to have control of the memory consumption. Both problems are addressed in [Wuy01] where Wuyts gives first insights on how he intends to solve these open issues.
4.3. SOFA

In this section we presented the PECOS component model which mainly aims to provide capabilities to define components and applications for a special kind of embedded systems, the so-called field devices. Like Fractal, gives PECOS component model the modeled components control over scheduling its sub-components and its memory consumption.

4.3. SOFA

In the category of platform-specific hierarchical component models the SOFA 2.0 provides the most capabilities. SOFA not only provides static capabilities of modeling component architectures. It also provides capabilities to specify the behavior of components and architectures through behavior protocols [PV02], similar to the Fractal component model. Beyond these static capabilities, SOFA assists in the deployment phase of an application as it supports the automatic generation of connectors that allow a seamless and transparent distribution of an application [BDH+08]. The SOFA component model is complemented by a distributed runtime environment.

4.3.1. Static View

The SOFA component model uses an ADL driven approach to describe the hierarchical components/component-based architectures. Its successor SOFA 2.0, instead, uses a meta-model driven approach [BHP06].

SOFA uses the generally applicable distinction of component views into black-box and white-box views. A gray-box view, as a third category, in SOFA represents a view on the internally visible structure of a component and its communication/relation with its sub-components [BHP06]. Based on SOFA's meta-model these are realized as instances of the Frame class (black-box component) and an Architecture class in case of a gray-box view of a component [BHP06, BDH+08]. The relations between sub-components' and between the encapsulating component
and the sub-components are realized through connectors [BDH+08] as instances of the Connection and Endpoint meta classes.

Like Fractal, runtime reconfiguration plays a central role in SOFA. Reconfiguration patterns are used to model reconfiguration capabilities. The factory pattern is used to add new components to the architecture at runtime, the removal pattern instead is used to remove components from an architecture during runtime. Finally, the utility pattern has been introduced to allow access to a component’s interfaces. SOFA uses the terminology of provided and required interfaces, across the component’s boundary [BHP06].

4.3.2. Behavior View

Behavior protocols [AP05a] are used to describe the behavior of software components as a “set of traces of events appearing on the component’s interfaces” [BDH+08]. In SOFA’s enhancement towards version 2.0, the behavior protocols have been extended to the Extended Behavior Protocols (EBP). Synchronization of multiple events and enumeration of data types have been added in the EBP.

A combination of event tokens, operators of regular expressions and a parallel composition operator are used as the basis of SOFA’s behavior protocol [BDH+08]. As method parameters and local variables have become part of the EBP, it is possible to describe the behavior of a component more precisely. Further details on EBP are found in [Kof06], where further details on the former behavior protocols are found in [AP05b].

4.3.3. Deployment View

In addition to the static description of components, their relationships to other components and their behavior, SOFA 2.0 provides a runtime environment, which is called SOFANode. This runtime environment consist of a repository where the component’s meta-data and code is stored and a set of deployment docks in which components are instantiated and executed.

Like Fractal where the concept of a controller is used as boundary of a component and its environment - invocations of the component’s provided functionalities are
only possible through the controller - SOFA uses a similar concept. In SOFA, components are regraded as being divisible into a functional and a control part. The functional part is thought of as representing the pieces of functionality of a component which are either provided directly by the component or provided through a composition of sub-components. The controller part, instead, is used to control non-functional properties of the component, such as lifecycle management and bindings [BDH+08].

The lifecycle of a SOFA application starts with the implementation and/or composition of components and their storage in the SOFANodes repository. In an assembly phase the composition hierarchy of an application is defined by starting with the top-level component and recursively solving the dependencies of its sub-components, sub-sub-components and so on, until the whole architectures dependencies are solved. At this stage the application can be deployed and executed in the SOFANode environment, where additional preferences need to be set, such as the control aspects that should be observed. The deployment information, however, is stored in a deployment plan which is used as the basis for the execution in the runtime environment.

### 4.3.4. Performance View

The deployment plan is used as basis of the performance view. However, quality of service (QoS) statements about an application can not be directly derived from a deployment plan and a separate performance model is introduced. The aim of this performance model is, however, not to statically describe the QoS as it is done in QoS description languages like QML [AE02] or CQML [ZM03], but to predict the performance of an application.

An iterative approach is used by SOFA’s performance model to predict the application’s performance. In this approach, shared components form the basis for calculating the performance of particular QoS attributes. These are used by the performance model to predict the performance of the higher-level components.
4. Frameworks and Methodologies

4.3.5. Conclusion

SOFA is currently the most powerful framework with an underlying platform specific hierarchical component model. It provides capabilities for modeling components and component composition by using a meta-model driven approach in SOFA 2.0, where the former SOFA model makes use of an ADL driven approach. Like PECOS it provides capabilities to specify the component’s behavior. Furthermore, SOFA provides a runtime environment (SOFANode), that allows SOFA applications to be deployed and executed. A performance model completes this framework for component based systems, by providing a performance prediction of applications.

In KobrA the separation of projection/views is a central aspect of the development methodology where the defined views - structural, operational and behavioral - are used to highlight the different facets of a single component and its associated elements. In SOFA views are used to highlight the different facets of a complete component composition representing a particular application.

4.4. KobrA

In the last chapter we identified two major categories of component models with an encapsulating framework or methodology for the development of hierarchical component based systems: the platform specific and the platform independent component models. The latter, however, has further been refined and a sub-category - the view centric component models - has been identified. KobrA’s underlying component model is the only one being both view centric and platform independent.

The acronym KobrA stands for the term “Komponenten-basierte Anwendungsentwicklung (KobrA)” – German for “Component-based Application Development”. The basic goal of KobrA is to allow the design of a complex system/component to be split into separate parts which can be tackled separately and in an hierarchical way. The KobrA component model aims to provide capabilities to describe a complex system (that as in PECOS, is interpreted as being the component at the root of a composition hierarchy) on the design level, where it makes heavy use of the diagram types provided by the UML. However, KobrA is not specific to the use of UML and any other language can be used to model a system as well. In [ABB+02]
Atkinson et al. recognized that for the development of a complex systems it is best to treat the different facets of a system, by separating the different concerns in a clean way.

### 4.4.1. Separation of Concerns

KobrA was mainly developed to cope with the issues arising from product line engineering, component based development and model driven development [ABB+02]. The “separation of concerns” principle is identified to be a central aspect for managing and developing complex (not necessarily component based) systems. KobrA makes use of several sub types of this basic principle. These are presented in the following.

#### 4.4.1.1. Separation of Development Dimensions

In KobrA it was recognized that the complexity of a large system is only manageable if the development dimensions are separated. The three dimensions are:

- **abstraction** which filters out information that is not needed at a particular place or point of time,
- **composition** which describes how a large complex artifact is made up from smaller and simpler parts and,
- **genericity** which captures information across families of similar artifacts.

These dimensions, illustrated in figure 4.2, are one of the foundations for defining viewpoints and the goals of development activities.

#### 4.4.1.2. Separation of Projections/Views

The KobrA component model makes use of different views to describe a component’s content. It strictly separates three fundamental information projections/views:

- **Structural projection**: The structural view of a component describes the classes and relationships which the subject of the view is involved in and (depending on the encapsulation level) its structural composition.
4. Frameworks and Methodologies

**Operational projection:** The operational view describes the functions (i.e. operations) that the subject of a view possesses and (depending on the encapsulation level) any functional decomposition that these functions participate in.

**Behavioral projection:** The behavioral view describes the timing and algorithmic properties of the subject of the view and (depending on the encapsulation level) any behavioral composition that the subject participates in.

As illustrated in figure 4.3, these are orthogonal to one another and are applied uniformly across all three development dimensions identified in the previous subsection.

KobrA defines various “localized” views on the system based on its principles of locality and parsimony, i.e. a view contains only the current component under con-
4.4. KobrA

consideration – the so-called subject – and its immediate environment. The properties of the subject are then described from the perspective of these three projections. In figure 4.3, the cloud represents the underlying component that is the current subject of development and the surfaces at right angles represent the three different projections. However, in the current enhancement of KobrA towards version 2.0, we recognized that additional dimensions need to be taken into account such as a dimension concerning non-functional properties.

4.4.1.3. Separation of Specifications and Realizations

One of the oldest “separations of concern” principles in software engineering is the principle of separating the description of “what” something does from the description of “how” it does it. KobrA refers to the former as the specification of the component and the latter as the realization.

The realization describes its internal architecture – what it is composed of, how it makes use of other components and what internal data structures and algorithms it uses. The specification, instead, contains the subset of the information in the realization which is intended to be externally visible to clients. In other words, the specification defines the overall interface to the component. It defines the interface(s) of the component in terms of the possible messages that it can support (i.e. the lists of operation signatures that it supports), but also contains additional behavioral, operational, and structural information. The specification also defines the component’s contracts with each of its clients.

As illustrated in figure 4.4, the separation of the specification from the realization is applied across all projections uniformly (i.e. it is orthogonal to them). In figure 4.4 the inner, slightly darker area of each projection depicts the specification part, while the outer area depicts the realization. The intent is to show that the specification should be thought of as a subset of, or a window onto, the realization.

4.4.1.4. Separation of Process and Product

The fourth fundamental separation of concerns in KobrA is the separation of product issues – the question of “what” should be built – from process issues – the question of “how” and “when” these things should be built. This is achieved by defining
4. Frameworks and Methodologies

![Figure 4.4.: Separation of Specification and Realization](image)

The artifacts making up a KobrA representation of a system separately from the activities and guidelines used to create and maintain them. In other words, the arrangement of KobrA views and their relationship are defined completely independently of the notion of time. In effect, the different projections of a component depicted in figure 4.3 represent a snapshot of the component at a particular point in the development process.

The advantage of defining the views of the system independently of process concerns is that they are completely agnostic to the process used to develop them. In other words, KobrA allows the information in each of the fundamental dimensions defined above (abstraction, composition, and genericity) to be elaborated in any order. Thus, in the composition dimension it is possible to develop composition information in a top-down way or a bottom-up way. In the abstraction dimension it is possible to develop abstraction levels in a forward engineering or a backward engineering way, and in the genericity dimension it is possible to develop specialization levels by specializing from the general, or generalizing from the specific. Of course, in practice, a combination of the extreme approaches is usually applied.

A key principle in KobrA is that the system and all components of the system should be treated alike (i.e. viewed in the same way) and that the composition hierarchy should be made explicit. As illustrated in figure 4.5, therefore, in general the complete description of a system is composed of a nested hierarchy of components, each of which can be viewed using the same projections outlined above. Of course, there are strict consistency constraints between the views of composite components and their sub-components and client components and their server. The properties of a
sub-component as represented in its specification have to match the usage of the component as defined in the realizations of its clients and the component it is contained within. As can be seen from figure 4.5, in UML terms, a KobrA representation of a component-based system can be thought of as a nested tree of component views (i.e. specifications) and composite system views (i.e. realizations).

4.4.2. Hierarchical Composition

Typically, the functionality provided by components on a lower level of a composition hierarchy are more specialized in comparison i.e. to its direct successor in a composition hierarchy. A “top-level” component is thought of as representing the application’s provided functionality in combination with its environmental dependencies. KobrA allows the application and its environment to be defined in a specialized view - the context realization - which can be thought of being a starting point of the development of an application in a top-down approach.

After initial steps in the hierarchical development process, where the system’s context is modeled, a component representing the system is modeled as the root of the composition hierarchy. In KobrA the subject’s specification depends on the realization level of the component it refines. In case of the root component the context realization is used as basis for the definition of its specification [ABB+02].

The subject’s provided operations and dependencies to other components that are modeled in its specification are refined in the subject’s realization. The pieces of
4. Frameworks and Methodologies

functionality covered by the subject’s operations are refined by dividing them in sub-parts each of which represents a particular piece of the original functionality. The pieces of functionality are grouped and sub-components are identified/defined, these pieces of sub-functionality are assigned to. However, beside assigning pieces of sub-functionality to sub-components, some of the pieces of sub-functionality are covered by the subject itself. In Figure 4.6 we give an example of the decomposition of one of the subject’s provided operations, how it is performed in the behavioral view of the subject’s realization.

![Figure 4.6. Abstract realization of one of the subject’s operations](image)

After decomposing the subject, several sub-components are identified each of them covering particular pieces of the subject’s responsibilities encapsulated in particular operations. The context of a particular sub-component is defined in the realization level of its “father” component. The sub-components are modeled in the same way as the subject, by defining a specification and a realization. In KobrA’s hierarchical refinement process the sub-components are then recursively decomposed till a particular level of granularity is reached.

### 4.4.2.1. Leaf Components

However, a component does not necessarily need to define a realization. There exist mainly two cases where only a specification is defined. In the first case the considered component is a leaf component that can not be further refined. Mostly, leaf-components have a realization as well. Only in a small number of cases do
4.4. KobrA

leaf-components not depend on their acquired components or do not define internal operations, so that a realization level is not necessary. Usually, the provided operations make use of the operations of the subject’s acquired components or internal operations. In this case a realization level is necessary to fully describe a leaf-component. In the second case the component should not further be refined as for instance a COTS component is used to fulfill functionalities defined in its specification [ABB*02].

4.4.3. Component Types

In the structural view of the subject’s specification the structural relationships between the subject and its associated components are modeled. As the specification defines a black box representation of the subject, in its structural view only those components are modeled that need to be provided to the subject. The associated components provide operations that are used by the subject’s operations to fulfill their contractually specified obligations. From the subject’s perspective it acquires these components (see Figure 4.7). It should be noted that in the structural views - specification and realization - beside components basic entities are modeled. In KobrA entities can easily be identified as they are allowed to have several attributes but do not provide any operation (except getter/and setter operations).

![Figure 4.7.: Abstract Example for a structural view in the specification](image)

In a subject’s realization the result of its local refinement process - also called decomposition - is modeled. In a subject’s decomposition sub-components are identified that provide pieces of the subject’s operations. In the structural view of the realization the subject and its associated components are modeled. The com-
4. Frameworks and Methodologies

Components that are modeled in the specification level are associated to the subject through «acquired» associations. To highlight that the pieces of functionality provided by the subject are based on the composition of the operations provided by its sub-components, the subject and its sub-components are associated by use of the composite aggregation. The association between subject and its sub-components is further described by use of the «contains» stereotype (see Figure 4.8).

![Figure 4.8.: Abstract Example for a structural view in the realization](image-url)

Different stereotypes have been used in the past to further describe the relation between the subject and its sub-components. In [ABB+02] the «create from» stereotype is used to highlight the composition between the subject and its sub-components. In [ABB+08] the associations are further described by use of the «made from» stereotype. Finally, [FA08] uses the «required» stereotype to highlight this association type. Anyway, from the subject’s perspective it contains the sub-components. Thus, the «contains» stereotype describes best the special kind of associations between the subject and its sub-components.

4.4.4. Conclusion

KobrA mainly aims to provide capabilities to model a complex system in a structured way. The separation of concerns plays a central role in KobrA. Like PECOS, KobrA considers the application/system as the top-level component of the composi-
4.5. Comparison

The externally invokable functionality of the modeled system is provided through the root of the composition hierarchy. The operations of the root component usually invoke the specialized operations of its sub-components to fulfill its contractually specified obligations. Anyway, not all of the functionality covered by the modeled systems is usually provided by the components of the composition hierarchy and external components are used to cover particular issues. Therefore, KobrA distinguishes between two different component types - the contained or sub-components and the acquired components.

The other frameworks and methodologies with an underlying hierarchical component model that are presented in this chapter basically describe what needs to be taken into account when a component based system is developed for a particular domain. KobrA, instead, has its focus on the modeling process and effectively allows to model a component based system in a hierarchical and view based manner.

4.5. Comparison

In [RRMP08] the component models and their encapsulating frameworks or methodologies have been applied to a general problem - the development/modeling of a trading system as can be observed in a supermarket handling sales - to allow a comparison between these. Unfortunately, due to the focus of each of the presented frameworks/methodologies, an objective comparison amongst them seems not to be possible.

However, in all of the presented component models the principle of “separation of concerns” is found. Fractal, PECOS and SOFA make use of an ADL or at least an ADL-like description for the composition of components. KobrA as a platform independent component model, instead, makes use of the structural view (e.g. UML class diagrams) to describe the composition hierarchies. The behavior properties of a component composition and the interaction amongst components and sub-components are mainly described by using a specific description language. In terms of KobrA these properties cover both the functional and the behavioral aspects of a component and its composition. Thus, it can be modeled by the use of the respective modeling diagrams (e.g. activity, sequence, collaboration diagram). To cover the specialized capabilities of the presented platform specific component
4. Frameworks and Methodologies

models, use of the UML can be made as well. E.g. the control capabilities of Fractal components are inherited from a particular interface which can easily be modeled in KobrA.

Different component types are used to describe the responsibilities of specialized components in some of the analyzed component models. Fractal, PECOS and SOFA make use of the assignment of responsibilities to special kinds of components. The specialization of components such as the passive components in PECOS are mainly related to the basic functionality this special kind of component provide. Related to the composition of components to create “complex” components the specialization has almost no influence on the relations allowed. Restrictions are mainly found in only allowing a subset of the possible relations to other components. Therefore, these restrictions, due to the specialization of component types, do only play a subordinate role for the development of a commonly applicable view on the composition of components.

In contrast to the other frameworks and methodologies that are mainly realization centric, KobrA’s aim is to provide a methodology to model complex hierarchical component based systems. Basically, KobrA is able to cover the special issues of the other component models to a certain degree. The different kind of component types defined in PECOS, Fractal and SOFA are specialization of the component types used in KobrA. The shared components defined in Fractal, for instance, that are accessible throughout the application are basically sub-types of the acquired components defined in KobrA. As acquired components are defined individually for each component, the shared component are the acquired components defined for the root-component in a composition hierarchy. Components having introspection or configuration/reconfiguration capabilities basically define a special behavior. This can easily be adopted in KobrA by defining specialized operations that allow to access these capabilities defined in Fractal. How a Fractal component takes control of its encapsulated components, however, can basically be described by use of KobrA’s behavioral view in a component’s realization.

The different kind of component types defined for PECOS are primarily based on the control capabilities these components are defined to have in a particular application. From KobrA’s perspective the component types defined in PECOS define a particular behavior that can basically be modeled in the behavioral view of a com-
4.5. Comparison

ponent’s specification (states and state transitions) and its realization (invocations of operations provided by the subject or other components). The composition defined in PECOS by use of the CoCo is covered by the hierarchical refinement of a component’s responsibilities in KobrA.

KobrA is also able to cover the special issues of SOFA. The black-box and white-box abstraction used to distinguish the details available for particular components, is directly covered in KobrA. Components are defined to expose exclusively their specification (black-box view) or both specification and realization (white-box view). The gray box categorization in KobrA basically addresses the details available in a heterogeneous branch of the composition tree. The component’s behavior that in SOFA is defined by use of behavior protocols is covered by the behavioral view of a component’s realization. The behavioral view defines how the subject’s provided functionality is realized by making use of operations provided by other components. How these in turn realize their provided functionality is described in the definition of the respective component due to KobrA’s principle of locality. The deployment and the performance view defined in SOFA do not directly influence the dependencies between the components and their underlying elements in a composition hierarchy.

Therefore, from a modeling perspective KobrA almost directly but at least indirectly covers the special issues of PECOS, Fractal and SOFA.
In the previous chapter we presented an overview of commonly used component based frameworks and methodologies with an underlying hierarchical component model. The composition hierarchies defined by these component models are based on the instances of components and related elements. KobrA is an exception as it additionally allows to define composition hierarchies based on component types. However, there is a fundamental difference between instance composition hierarchies and type composition hierarchies. This difference has a major influence on the purpose of defining composition-hierarchy-aware software metrics.

We start this chapter, with a detailed discussion on instance and type composition hierarchies. These, however, require the compliance with particular rules for composition that are presented afterwards. Based on the rules and the two different kind of composition hierarchies, we define a hierarchical component model building the basis for the definition of hierarchy-aware software metrics.

5.1. Composition Hierarchies

Reflecting on the different kind of hierarchical component models that were presented in the previous chapter, we determined that these basically address compo-
5. Hierarchical Component Model

Composition hierarchies defined by component instances. An exception is made for KobraA that additionally allows to define composition hierarchies based on component types in a structured way. The difference between instance and type hierarchies is explained in the following.

5.1.1. Instance Hierarchies

Without referring to a particular hierarchical component model, we start our discussion on instance composition hierarchies with a simple example. We use the simple model of a mail server that consists of three components: the MailServer, the UserDatabase and the BlackListServer component. We define the UserDatabase to be an integral part of the mail server, where it is contained in. The BlackListServer, instead, is acquired by the MailServer and needs to be associated to it, enabling the MailServer to fulfill its contractually specified obligations.

From the composition perspective this means that the UserDatabase component is contained in the MailServer component. In other words the MailServer component is composed of the UserDatabase component. An instance of the BlackListServer, instead, needs to be associated to an instance of the MailServer during execution. Thus, it is acquired by the MailServer component. The relationships between these components are illustrated in terms of the UML composition diagram in figure 5.1.

![Instance Composition Hierarchy: Mail Server with UML component diagram](image)

Although we defined the relationships between the components based on their type definition, it is the relationships among the instances of these types that set up the composition hierarchy. In the following we define a particular instance of the component based system. First, we assume an instance of the MailServer to be a university's mail server. The contact mail addresses of the university's students and employees are hosted by two distinct servers. Further, a central black list
5.1. Composition Hierarchies

A component is hosted by a central server. The relationships among instances of the components are illustrated in figure 5.2 by use of the UML object diagram.

Figure 5.2.: Instance Composition Hierarchy: University Mail Server

To highlight that the instances of the user database are contained within the instance of the mail server, we defined the relations to be \textit{i-contained}. \textit{I-contained} relationships are a shortcut for \textit{instance contained}, which means that the instances of a component are contained in the instance of another component. Further, the instance of the BlackList component needs to be provided to the university's mail server to fulfill its obligations. To highlight that it is associated to the mail server and not contained in an instance of it, we defined the association to be \textit{i-acquired}, which is a shortcut for \textit{instance acquired}.

\textbf{Definition} (I-Contained Relationships). \textit{Instance contained relationships, or \textit{i-contained relationships for short, are defined between instances of components, where an instance of a component is contained within another.}

\textbf{Definition} (I-Acquired Relationships). \textit{Instance acquired relationships, or \textit{i-acquired relationships for short, are defined between instances of components, where an instance of a component requires an instance of another component to fulfill its contractually specified obligations.}

The instance composition hierarchy is defined by the \textit{i-contained} relationships among components. However, in software modeling generally the types in terms of UML, classes or components, are used to determine the relationships between the instances. At the beginning of this section we used a UML component diagram to highlight the relationships. Being consistent with the object diagram describing the relationships of the university mail server we generalize the relationships between the components by use of a UML class diagram, where classes are used as representatives of components enhanced by a Component-stereotype (see figure 5.3).
5. Hierarchical Component Model

**Figure 5.3.:** Instance Composition Hierarchy: Mail Server Composition Hierarchy

**Definition** (Instance Composition Hierarchy). *The instance composition hierarchy is defined on basis of the i-contained relationships between component instances.*

The instance composition hierarchy has a component instance at the top of the hierarchy in which component instances are i-contained. These, however, can i-contain component instances by their own, that by theirselves can i-contain component instances and so on. In figure 5.4 an instance composition hierarchy is illustrated schematically.

**Figure 5.4.:** Instance Composition Hierarchy

### 5.1.2. Type Hierarchies

Basically all of the hierarchical components models presented in the previous chapter are instance hierarchies. However, in the KobrA methodology and underlying
5.1. Composition Hierarchies

hierarchical component model [ABB+02], Atkinson et. al. also discuss the logical relationships between component definitions - the component types.

Physical containment relationships of component types are affected by principles like private containment as it is used by several object oriented programming languages. In Java, for instance, private containment is given by class definitions being defined in another class definition. However, Java embraces also a second kind of containment relationship that is based on the physical containment of classes in a package structure. The packaging style of Java does not allow classes to be part of two concrete packages at the same time. For example a class X can not be contained in package a.b.c and at the same time in a.b.d. The UML allows the definition of packaging based containment relationships by use of UML package diagram.

Referring to the example introduced in the previous section, we discuss the component type relationships in the following. We start our consideration with the MailServer-component. At this point we assume that the definition - the component type - of the UserDatabase is defined within the MailServer component. The containment relationship among the MailServer and the UserDatabase-component types - the one is defined within the other - is different from the one of the instances that are discussed in the previous section. Therefore, we introduce a type-contained relationship - t-contained for short - among component definitions.

Further, we assume the BlackList-component not to be part of the application and that it is defined somewhere else. This relationship is called analogously t-acquired. In object orientation t-acquired relationships among classes are found as well, where, for instance, a class being defined in an external library is used in someone's own application. In figure 5.5 the type relationships for the mail server example is illustrated.

**Definition (T-Contained Relationships).** Type contained relationships, or t-contained relationships for short, are defined between the definitions (e.g. source code representations) of component types, where the definition - the type - of a component is contained within another.

**Definition (T-Acquired Relationships).** Type acquired relationships, or t-acquired relationships for short, are defined between the types of component, where a component type acquires another component for its definition.
5. Hierarchical Component Model

Based on the containment definition of component types, a composition hierarchy is defined on these. Like the instance hierarchy component types can contain other component types that in turn can contain other component types and so on, which results in a type containment hierarchy.

Definition (Type Composition Hierarchy). The type composition hierarchy is defined on basis of the t-contained relationships between component types.

5.1.3. Meta-Level Relationships

In the previous sections we introduced the instance and the type hierarchy more or less independently of each other. But by referring to meta-modeling, different meta-levels are addressed by the component type, the components and instances of it. Strong dependencies between both kind of hierarchies exist from the meta-level perspective. These are discussed in detail in this section.

Several meta-modeling approaches use the term “meta” in an absolute sense to describe the level of a model and their elements [Atk97, p.94]. Different meta-modeling levels (most approaches limit themselves to four levels) are defined. For example the meta-object facility (MOF) deals with the data ($M_0$-level), the model ($M_1$-level), the meta-model ($M_2$-level) and the meta-meta model ($M_3$-level). The $M_0$-level depends on the $M_1$-level, where it is an instance of. The $M_1$-level in turn is an instance of the $M_2$-level, which in turn is an instance of the $M_3$-level. The relationships among these levels are illustrated in figure 5.6.

Referring to the previous example the universityMailServer ($M_0$ level) was introduced as an instance of a MailServer ($M_1$ level), which in turn is an instance of an
element of the $M_2$ level. This in turn means that the MailServer has two facets. First, it defines a template for the $M_0$ level instances. On the other hand, it is an instance of a $M_2$ level element, as well. To highlight that an element of a particular level $M_x$ is both, template of an $M_{x-1}$ level element and instance of an $M_{x+1}$ element Atkinson [Atk97] introduced the concept of clabjects (CLAss and oBJECT). In a clabject the term class is used synonymously for template and object synonymously for instance. In figure 5.7 both facets of the $M_1$ level of the mail server example are shown.

We ignore the different kind of associations between the elements of the $M_1$ level for a moment. In the template model (left hand side of figure 5.7) the relationships between the three types MailServer, BlackListServer and UserDatabase are illustrated. From a clabject perspective the right hand side of figure 5.7 models the instance view. All types are instance of the meta-class ComponentType.
In the template view of the $M_1$ level the instance contained/acquired relationships are defined. These are instantiated at the $M_0$ level. In other words, we define the i-contained/acquired to exist at the template view of the $M_1$ level and the instance view of the $M_0$ level (see figure 5.8).

The type-contained/acquired relationships, instead, are defined for the instance view of the $M_1$ level and the template view of the $M_2$ level. The relationships between the component type instances are illustrated at the left hand side of figure 5.9, where the template view of the component types are illustrated at the right hand side.

5.1.4. Instance vs Type Relationships

Based on the definition of the different meta-levels we defined that the instance relationships (i-contained/acquired) are defined for the instance view of the $M_0$ level and the template view of the $M_1$ level. The type relationships, instead, are defined for the instance view of the $M_1$ level and the template view of the $M_2$ level.
However, up to this point we did not go into detail on the dependencies between instance and type relationships. We defined that the type relationships are based on the containment rules of the type definition. In terms of object orientation a class that is defined in another is type-contained in it. A different kind of type containment is defined by the package structure. However, we do not assume any particular kind of type containment at this point, we only define that type containment exists (see right hand side of figure 5.9). The definition of a type containment implies the definition of acquired types. These basically can be interpreted as of being the opposite. In terms of object oriented programming components or elements of external libraries can be interpreted as of being type acquired.

Instance relationships, instead, are based on the containment rules of type instances. Referring to object orientation a particular kind of i-contained relationship is given by instances (objects) creation within an operation. An example of i-acquired relationships is given by the parameters that can be objects, instances of components or software entities and so on. These are not available nor are these created by the component under consideration and need to be provided, when the component instance is used.

Anyway, there is a strong dependency between the instance and type relationships. Both, i-acquired/contained can be instances of t-contained/t-acquired relationships.

5.1.4.1. I-contained vs T-acquired/contained relationships

Basically i-contained relationships can either be instances of type-acquired or type-contained relationships. For example by assuming the creation of a component X within the component under consideration, X can either be t-contained in terms of being defined within the component under consideration or being t-acquired if it is defined somewhere else. This implies for the mail server example that the UserDatabase component that is i-contained in the MailServer component can either be t-acquired or t-contained in the MailServer component, where in the given example it is defined as a t-acquired relationship (see template view vs instance view of the $M_1$ level).
5. Hierarchical Component Model

5.1.4.2. I-acquired vs T-acquired/contained relationships

An i-acquired component relationship can also imply a type-acquired or type-contained relationship. At a first glance the i-acquired/t-contained relationship does not make much sense if we think of private t-containment. A component that is privately defined in another component usually can not be instantiated from a different component. However, if we assume public containment, instead, a component being i-contained can be instantiated by other components as well. This implies that it can be i-acquired by the component under consideration. Anyway, the normal case is defined by the i-acquired/t-acquired relationship among components.

In the mail server example, the MailServer-component has an i-acquired relationship with the BlackListServer-component (see template view of the $M_1$ level). Further, in the instance view of the $M_1$ level it is defined to have a t-acquired relationship with the BlackListServer-component type. Thus, there exists an i-acquired/t-acquired relationship.

5.2. Composition Rules

Composition hierarchies are a fundamental principle of hierarchical component models. In PECOS, for instance, the root of the composition hierarchy is used as representative of the application. The provided operations of the components in the underlying instance composition hierarchy are used to fulfill the obligations of the application's contractually specified operations. KobrA uses a similar concept.

Anyway, at this point we set up some simple rules affecting the acquired relationships of instance and type hierarchies. At a starting point of our discussion we define the root of a composition hierarchy to be a representative of the application. Its contained relationships, the contained relationships of the root's contained components and so on are setting up a composition hierarchy like the schematic one illustrated in figure 5.10. In figure 5.10 the instance containments are used to set up the hierarchy.

From the perspective of the component instance of $C$, it directly i-contains instances of the components $F$ and $G$. As these by itself i-contain instances of components e.g.
5.2. Composition Rules

the instance of $F$ i-contains the instance of $I$, the root of the instance tree indirectly i-contains the instance of $I$.

Basically all of the component instances in the composition hierarchy can have i-acquired relationships to other component instances. For example the instance of component $I$ can have an i-acquired relationship with the instance of $J$. Although both instances are contained in the composition hierarchy of the instance of $C$ and the instance of $F$, an i-containment relationship between both instances does not exist. As the instance of $I$ can acquire an instance of $J$ to fulfill its obligations, it could i-acquire the instance of $J$ residing at the same level of the composition hierarchy (see figure 5.11).

Without loss of generality we define that in hierarchical composition hierarchies instance and type acquired relationships can only be defined between nodes (in-...
5. Hierarchical Component Model

instances or types) that are not on the same path from the root of the composition hierarchy to the node being considered. This implies that for example the instance of F in the composition hierarchy defined in figure 5.10 could have i-acquired relationships to instances of G and all of its i-contained instances (K and L).

We further restrict this path rule as we define a rule which induces some kind of disciplined composition hierarchy. A directly contained component of X is only allowed to acquire components (instances or types) which are directly contained in X or components that are acquired by X. Unfortunately, by applying this rule on a composition hierarchy means that the root of the composition hierarchy could not have any acquired relationships. The root is not contained in any other component and thus there is no component it could “inherit” acquired components from.

To tackle this issue, we enhance the composition hierarchy by a single level. A new root node, some kind of virtual node is introduced. All acquired components being external to the composition hierarchy are related to the virtual root. The former root is directly related to the virtual root as a contained component. Thus, the root can acquire “external” components following the disciplined composition rule via the virtual root.

In figure 5.12 an example of a virtual composition hierarchy is illustrated. An instance of component A is newly introduced as root of the virtual composition hierarchy. This “virtual” component instance is used to keep track of the externally acquired nodes. The core composition hierarchy with the anonymous instance of C being the root is directly integrated in the virtual composition hierarchy. Based on the disciplined composition hierarchy rule the instance of C is only allowed to have an i-acquired relationship with the instances of the external components (grayed out) B and D.

The external nodes that are acquired by instances of the core composition hierarchy are associated to the newly introduced root node. In figure 5.12 the i-acquired instances that are external to the core composition hierarchy are grayed out. For example, the instance of L has an i-acquired relationship to the instance of D that is not part of the core composition hierarchy. Due to the disciplined composition hierarchy rule L can only i-acquire the instance of D if it is directly i-contained in the component L is contained in or that L’s parent (the instance of G) in the composition hierarchy also acquires the instance of L. As the instance of G unfortunately is not
an immediate peer of the instance of $D$, it needs to be defined by as i-acquired by $G$'s parent. The instance of $G$'s, however, is the root of the core composition hierarchy and the external acquired components ($B$ and $D$) are immediate peers of the root. Thus it can have an i-acquired relationship to this.

If we invert the considerations, $L$ can only acquire a particular external component if all of its predecessors in the hierarchy acquire this external component as well. Anyway, let’s consider the possible i-acquired relationships of the instance of $F$ in the composition hierarchy illustrated in figure 5.12. First, it is allowed to have an i-acquired relationship to the instance of $G$ as it is an immediate peer. Second, it is allowed to have i-acquired relationship to the external component instance $D$ as its parent i-acquires $D$.

A third rule - the Inter Invocation Rule - is introduced focusing on the interactions between the operations encapsulated in components that are affected in the invocation chain of an operation. We explain the Inter Invocation Rule by use of the following example: the operation $opA$ that is encapsulated in the instance of ComponentA calls the operation $opC$ that is encapsulated in the instance of ComponentC. The operation $opC$ invokes the operation $opB$ of ComponentB. ComponentC i-acquires ComponentB. Finally, $opB$ invokes the operation $opD$ encapsulated in ComponentC (see figure 5.13).
5. Hierarchical Component Model

It is very unlikely that an operation calls an operation of another component that calls back an operation of the first component. Such a relation results in a high coupling between the affected components. From our point of view such relations between components should be avoided in component based development. Therefore, we define the Inter Invocation Rule that does not allow “component call backs”. In other words, an operation of a component X that is called by the operation of another component Y is not allowed to invoke operations of the component Y.

An overview of the rules that are assumed for the further consideration of composition hierarchies are given in the following. Although, we discussed the issues related to an instance composition hierarchy and i-acquired/i-contained relationships, these are also valid for the type composition hierarchies and t-acquired/t-contained relationships.

**Virtual Root Rule:** A virtual root of the composition hierarchy is introduced that tackles with all of the instances/types that have an acquired relationship to instances/types outside the core composition hierarchy.

**Disciplined Hierarchy Rule:** A component contained in a component X can only acquire components that are

1. also directly contained in X (immediate peers),
2. or that are directly acquired by X.

**Inter-Component Invocation Rule:** An operation of a component X that is called by the operation of another component Y is not allowed to invoke operations of the component Y.
5.3. Basic Model

In chapter 2 SOFTWARE COMPONENTS we gave an operation-centric definition of software components that is basically a combination of the definitions attributed to Szyperski [SP97] and Heineman and Council [HC01]. In this section we focus on the definition of a commonly applicable component model. Basically, the newly introduced component model is not tailored to any of the hierarchical component models that have been presented in the previous chapter. However, it can easily be adapted to these. It should be noted that it is not complete as it only addresses the issues that are of interest for the definition of hierarchy-aware software metrics that are discussed in the second part.

5.3.1. Component

As we have discussed earlier a component instance can have i-acquired and i-contained relationships to other components. Beside of being a “unit of composition”, a software component defines the contract between the component and its context through provided and required operations. In figure 5.14 we define the semantic relationships between operations and the component under consideration.

![Component Model: Basic Properties](image)

In practice software components are often thought of being organizational elements. In this case, components are interpreted as containers that associate its provided and required operations with internal elements - encapsulated by the component under consideration - that provide a realization for or acquire a particular
operation to fulfill their obligations. The realizations, however, can either be object oriented, procedural or can be based on any other paradigm.

In the literature, however, components are often thought as of being internally defined in an object oriented way. Even more often classes in terms of object orientation are interpreted as of being components as well. Basically we think that there is a difference between classes and components in general. In object orientation classes are defined to have provided operations, where they usually do not publicly define the elements they require. In contrast to classes, the specification of a component’s required elements is a central part of its definition. However, a class indirectly exposes its required elements as well.

In figure 5.15 the internal structure of a component is shown where a component internally can be defined as of being object oriented or procedural. Anyway, procedures can be interpreted as of being specializations of operations. Basically we assume that a component directly realizes its provided operations, e.g. by delegation to the operation of a contained class. Further, we define that a component can generally have attributes that can either be publicly visible or hidden within the component.

The attribute is defined to be of a particular type (Type-element). We define Class-elements and Component-elements to be specializations of a general Type. We further define DataAbstraction-elements as of being a specialization of Type. In contrast to classes, DataAbstraction’s are not instantiable and represent the encapsulated set of information that is defined for instance by the publicly accessible at-
tributes (including those being accessible through getter operations) within a class. We think that it is necessary to make a distinction between objects that are instances of classes and data abstractions at this point, as for instance the encapsulated information in modern technologies like web services are not passed through objects but are encapsulated in textual descriptions that are returned by the respective service (see figure 5.16).

5.3.2. Operation

Operations consist of several externally visible parts. An operation contains the definitions of pre- and post-conditions in the operation’s profile, the definition of the operation’s return type and the definition of its parameters’ types. Further, an operation has a body, in which the algorithmic part of the operation is defined (see figure 5.17).

In figure 5.18 the details of the semantic relationships of an operation’s body are illustrated. It should be noted that the component model presented in this chapter is not meant to be complete at all and basically has its focus on defining the relationships that are necessary for the development of hierarchy-aware metrics.

The semantic relationships of an operation’s body mainly refer to its basic elements - the statements. Statements consist of operators and operands. Anyway, we identified two different kinds of statement categories, the compound statements and the simple statements. In contrast to simple statements, do compound statements have a body by their own. These can either be different kind of loops, like do, for, for all
and so on, or branch statements like if...then...else. Simple statements, instead, are not allowed to have a body. These are for example method calls or assignment statements.

5.4. Conclusion

In this chapter we discussed the different kind of relationships between components in hierarchical component based systems. At the instance level of the modeling levels $M_1 \ldots M_4$, we identified that a dependency between the component instances based on the instance composition hierarchy exists. We categorized the different kinds of relationships among component instances as i-acquired/i-contained relationships.

Referring to the terminology of meta-modeling we further discussed the dualism of the elements in a particular meta-level. The i-acquired/i-contained relationships that are introduced on the instance part of the $M_0$ level, are also defined for the template part of the $M_1$ level. Due to the dualism of element descriptions in meta modeling, in almost each level there exists an instance part that is based on the template part of level above and a template part the instances of the level below are based on. In the instance part of the $M_1$ level we identified a different kind of relationship between component types. These are defined as t-acquired/t-contained relationships and are also defined for the template part of the $M_2$ level.
Based on the distinction of instance and type relationships we identified that two composition hierarchies exist. The instance composition hierarchy is the one most researchers think of when talking about hierarchical component based systems. On the other hand we identified the type composition hierarchy to be of major interest. The latter, however, has its focus on type definitions, like a component type being defined within another (t-contained).

Beside the definition of different kinds of acquired and contained relationships based on the modeling level, we discussed the relationships between these in detail. We realized that the instance relationships can be “inherited” from both kind of type relationships. Anyway, as we will discuss for hierarchy-aware metrics the different kinds of relationships have a major impact on the relation between metrics and quality characteristics.

As each component can acquire and require other components to fulfill its contractually specified obligations we introduced three rules addressing the characteristics of composition hierarchies. Due to the virtual root rule a virtual node is introduced at the top of the hierarchy. This virtual root is used to keep track of all of the acquired relationships to component (instances or types) outside the composition
hierarchy. Further, the **disciplined hierarchy rule** is introduced that clearly defines to which other components in the virtual composition hierarchy a component can have acquired relationships to. Finally, we introduced the **inter invocation rule** that defines that an operation of component X that is invoked by an operation of component Y is not allowed to invoke operations of component Y.

Finally, we defined a basic component model building the foundation for the discussion of hierarchy-aware metrics in the next part. The model, however, is based the insights gained on the component models and their encapsulating frameworks/methodologies discussed in the previous chapters.

Specializations of the basic meta-model towards the particular component models need to be performed. For instance, it would be necessary to enhance the basic meta-model to cover the issues of Fractal. The responsibilities covered by the specialized component types would need to be integrated in the model as well as the different kinds of capabilities of components.

Anyway, the basic model aims to provide a generally valid description of the semantic relationships between components and other elements in hierarchical component based systems. An overview of the component model is illustrated in figure 5.19.
5.4. Conclusion

Component Model

Figure 5.19.: Component Model
Part III

Measures
Generally, software metrics can be thought of being the basic entities of measuring software artifacts. However, metrics by themselves are of limited expressiveness unless they are interpreted in some way. A commonly used interpretation of software metrics is to interpret them as indicators for quality characteristics. However, other interpretations of metrics than mapping them towards quality characteristics are widely discussed in the literature as well, for instance, by interpreting metrics as indicators of complexity.

In the late 1970’s researchers started to analyze the dependencies and relationships between quality characteristics. Both, Boehm [BBL76] and McCall [MRW77], investigated the relationships and dependencies between quality characteristics resulting in the first quality models. The ISO/IEC 9126 [ISO91] is currently the most used and discussed quality model. The ISO/IEC 9126 was introduced in the early 1990’s and is closely related to the quality models of Boehm and McCall. Beside the definition of a general, hierarchical quality model for three stages of the developments process, a set of metrics is defined as indicators for the underlying quality characteristics. Basically, these are used to give a general indication of a particular
quality characteristic. Several specializations of the general quality model towards a particular domain are introduced in the literature, implying the need for defining a set of metrics addressing the purposes of the particular domain.

6.1. Indicator Metrics

Before we turn our attention to quality models, we briefly discuss the issues arising in an interpretation of metrics as indicators for particular quality characteristics. The mapping of metrics to be indicators of a particular quality characteristic is difficult. Take the metric Mean Time To Failure (MTTF) for example. The MTTF is commonly accepted as being a metric for reliability. However, a deeper look at the relation between metric and quality characteristic reveals that a metric would need to cover all of the facets found in the quality characteristic’s definition. But generally, this is not true since at the very most metrics only cover a particular facet of a quality characteristic. This implies that an indicator metric that is thought of as being able to cover all of the facets grounded on the quality characteristic’s definition would need to be based on a combination of metrics of the particular facets. An example of such a metric is given by Nagappan who combined several metrics to build a metric for assessing the reliability of software systems [NWVO05, Nag05].

However, the basic assumption that an aspect is fully covered by a particular metric is often made as well. Furthermore, there is another issue that needs to be taken into account. Although a particular metric might cover a facet of a quality characteristic, it still needs to be kept in mind that this metric might have an influence on other facets as well. This, however, makes it difficult for a combination of metric to fully cover a quality characteristic.

A commonly used way to investigate the mappings between metrics and quality characteristics is to use questionnaires as the basis for a poll of specialists. Bertoa et al. [BTV06] present a method based on questionnaires to investigate on how metrics indicate the usability of a software component based on Bayesian belief networks. The method presented, however, is generally applicable and can easily be applied to assess metrics for other quality characteristics as well. In [KG05],
Khosravi and Guéhéneuc, a further reaching method is introduced that consists of
nine individual steps to develop and validate a quality model and related metrics.
A similar method to introduce and validate quality models and related metrics is
introduced by the ISO/IEC SQuaRE project being an advancement of the ISO/IEC
9126.

6.2. General Quality Models

The quality models introduced by Boehm [BBL76] and McCall [MRW77] were the
first to describe the relationships and dependencies between quality characteristics
in a hierarchical manner. According to the definition of the IOS/IEC Std. 14598
[ISO99] we define a quality model as:

**Quality Model:** A quality model is defined as the set of characteristics and
the relationships between them which provide the basis for specifying quality
requirements and evaluating quality.

However, it should be noted that the terms quality factor, quality attribute and
quality criteria that are often used synonymously for the term quality characteris-
tic, are also used to characterize the different levels of characteristics in the hier-
archies. Boehm, for instance, used the terms quality factor and quality criteria to
classify the different levels of his quality model.

Since the presentation of McCall’s and Boehm’s quality model in the 1970’s, the
relationships between quality characteristics has been further investigated by sev-
eral researchers, resulting in particular quality models. The most well known ones
are the quality models by Firesmith [Fir03] and the quality model defined in the
ISO/IEC Std. 9126 [ISO91]. Dromey [Dro95] realized the lack of acceptance of
quality models in general and proposed a methodology for building a quality model
for a particular purpose. In [Dro95] he further evaluated his methodology on the
example of a component’s variables and expressions.

However, in industry and academia the ISO/IEC Std. 9126 [ISO91] is the quality
model most commonly used. It should be noted that all of the quality models make
use of a subset of the quality characteristics found in the literature. Khosravi and
Guéhéneuc, for instance, give a survey on quality characteristics and list more than
6. Quality Model - Related Work and Evaluation Processes

100 individual factors in [KG04]. These might be used to develop a quality model for a particular purpose.

6.2.1. Boehm’s Quality model

The quality model developed by Boehm in 1976 [BBL76] started with an initial set of 11 quality characteristics which extended to 24 quality characteristics during the development of the quality model. In the investigation of the quality characteristics the authors determined that a hierarchical relationship between some of the characteristic values exist. They came to the conclusion that the relation between the quality characteristics is represented best by a hierarchical quality model since they observed that, for instance, understandability as a quality characteristic has an impact on maintainability since “any code maintenance requires that the maintainer understand the code” [BBL76]. However, they also identified other characteristics of maintainability such as the testability that did not depend on understandability. The investigation of the relation between these characteristics resulted in a software quality characteristic tree that is presented in figure 6.1.

Based on the relation between some of the quality characteristics the authors of [BBL76] separate the characteristics into primitive and more general (high-level) characteristics, and use the term metrics as “measures of the extent or degree to which a product [...] possesses and exhibits a certain (quality) characteristic”.

We close the presentation of Boehm’s quality model by noting that the authors do not give any suggestion about how the characteristic values can be measured. But they came to the conclusion that “calculating and understanding the value of a single, overall metric for software quality may be more trouble than it is worth” [BBL76].

6.2.2. McCall’s Quality model

Like the quality model introduced by Boehm, the quality model of McCall was introduced in the late 1970’s. The model was developed for the US Air Force with the intention to bridge the gap between users and developers. Therefore, it reflects some kind of product view.
6.2. General Quality Models

The central idea, similar to the model of Boehm [BBL76], is to assess the relationships among external external quality factors and product quality criteria [MRW77]. The quality model combines eleven main criteria, where the primary goal was the identification of relationships between external factors, quality factors and metrics. The quality factors encompass the product operation (operational characteristics), product revision (ability to carry changes out), and product transition (adaptability to new environments). These criteria relate to the internal properties of software products. The quality model of McCall is illustrated in figure 6.2.

Each quality factor is thought of as representing a non directly measurable aspect of the software product’s quality linked with measurable properties (software metrics). He suggests to evaluate these properties by using charts ranging from 0 (goal not reached) to 10 (excellent realization). Further he realized that only a combination of weighted (c) metrics (m) can indicator the quality concerning a particular quality factor (F).
The weights, however, need to be determined by local organizational considerations. The relation between quality factors is influenced by many factors such as the domain, the technology used and the application that is going to be developed.

Thus, the model and as we see related to the research results of the last 30 years these models and the relation between metrics and particular quality factors
6.2.3. The ISO/IEC 9126 Quality Model

The ISO/IEC Std. 9126 Quality Model is the most well known quality model in the literature. The ISO/IEC 9126 was first developed in 1991 [ISO91] and was revised from 2001 to 2004. The ISO/IEC Std. 9126 breaks the overall quality of software entities into three views - the internal, the external and the quality in use - that can be assigned to different phases of the development process. Each of the views is characterized by a set of quality characteristics, which in turn are further refined by sub-characteristics [MBM+08]. The characteristics of the three views are attributed as internal, external and in-use quality characteristics.

The internal quality characteristics are related to the early stage of a development process and the static representation is used to “measure” them. The external quality characteristics are related to the execution of the software entities. However, basically no assumption on the context of the software entity is made, which includes the absence of the user at this “measurement stage” as well. Typically, the testing phase is used to “measure” the external quality characteristics. The quality characteristics related to the third stage, the quality in use characteristics, assume the measurement of a software entity in their final context, including the user’s involvement in the execution process.

Generally, the ISO/IEC Std. 9126 assumes a strong dependency between the three stages on the basis of the stage’s quality characteristics (see Figure 6.3).

![Figure 6.3.: Relations between quality views in the ISO/IEC Std. 9126](image-url)

It is assumed that the quality characteristics of the “external measurement view” depend on those of the “internal measurement view”, as well as the external quality characteristics influence on the ones of the “quality in use measurement view”. The dependencies between the views are based on the assumption that having a high external quality, for example, would result in a high in use quality. This assumption does not need to be necessarily true in most situations, as a high external quality does not necessarily lead to a high quality in use [MBM+08]. This is especially the
case when an overall quality needs to be assessed and many contradicting quality factors need to be taken into account. Moraga et al. give the example of an Ferrari not being “[...] the best car to use to go to work if your job is social assistant in a deprived suburb in the outskirts of New York.”

The “quality in use view” is based on four characteristics - *effectiveness, productivity, safety* and *satisfaction* - which are not further refined by sub-characteristics. In contrast, each of the views -“internal” and “external view” - is based on six quality characteristics - portability, maintainability, efficiency, usability, reliability, functionality -, further refined by in total 27 sub-characteristics. The dependencies between characteristics of the “external” and “internal quality view” are defined as being the same and are shown in figure 6.4.

![Quality Model Diagram](image)

Figure 6.4.: The ISO/IEC 9126 Quality Model of the internal and external view

Beside the quality model refined in the three views - internal, external and quality in use view - the ISO/IEC Std. 9126 presents a set of metrics for each of the quality characteristics and sub-characteristics. However, it is not further defined to which degree the presented metrics give an indication of a particular quality characteristic or have an impact on other than the quality characteristic these are attributed to, nor it is defined in which context these metrics can be used at all for being indicators of particular quality characteristics.

### 6.3. Specialized Quality Models

Due to the generic form of ISO/IEC 9126, the quality attributes and the metrics defined as being indicators of these are basically too general to deal with the specific characteristics of COTS components [BV02] or hierarchical component models.
In this section we give an overview of recent research activities focusing on defining quality models appropriate for component based systems in general and COTS components in particular.

### 6.3.1. Bertoa and Vallecillo

In [BV02], Bertoa and Vallecillo, discussed the drawbacks of generic quality models and the necessity of tailoring the generic models towards a particular domain or paradigm as proposed by the ISO/IEC 9126. The goal, however, was the definition of a quality model for COTS components.

Addressing the terminology of the ISO/IEC 9126 Bertoa and Vallecillo identified the necessity to distinguish between internal and external metrics. Although, external metrics are more appropriate for COTS components due to their black box nature, they recognized that the internal metrics can not be discarded as they often reveal an indirect measurement of the external metrics.

As in [FA08] they identified that other quality characteristics such as price, license conditions and so on are of major importance when selecting a particular COTS component. Anyway, Bertoa and Vallecillo recognized that not all of the characteristics of a software product defined in the ISO/IEC 9126 are of importance for COTS components.

Beside the ISO/IEC 9126’s main (internal and external) characteristics portability and fault tolerance, the sub-characteristics stability and analyzability disappeared in the quality model for COTS components. Two new sub-characteristics - compatibility and complexity - were introduced. The meaning of some of the characteristics and sub-characteristics in the COTS quality model has changed compared to the definitions of the ISO/IEC 9126 [BV02]. An overview of the quality characteristics of the quality model for COTS components is presented in figure 6.5 (internal) and 6.6 (external).

Beside the definition of the quality model for COTS components the authors introduce a set of attributes and related metrics addressing the component lifecycle (internal) and their behavior during runtime (external). For a detailed overview of the attributes and related metrics, however, we refer to [BV02] for further information.
6. Quality Model - Related Work and Evaluation Processes

In [KG04], Khosravi and Guéhéneuc, introduce a quality model for design patterns. However, before they go into detail on a specialized quality model they give a detailed overview of current quality models and their underlying quality characteristics. Further, they give an overview of definitions used in the literature for 106 quality characteristics related to object oriented programming. Based on these definitions a contingency table is presented in which Khosravi and Guéhéneuc show the interrelationships among the quality characteristics.

For the quality characteristics being of major importance they further introduce an overview of commonly used metrics without going into detail on the underlying model these metrics are based on. Based on the detailed considerations of quality models, quality characteristics and related metrics Khosravi and Guéhéneuc turn their attention to the definition of a quality model for design patterns.

Gamma stated that design patterns in general are commonly used solutions for specific design problems. The application of these generalized solutions are thought of as increasing the software application’s reliability, reusability, but above all the
application’s maintainability. Based on Gamma’s assumption on the benefit of applying design patterns Khosravi and Guéhéneuc identified a set of main characteristics as being appropriate for these (usability, reusability, flexibility, scalability and robustness). For a refinement of the main characteristics, sub-characteristics are introduced according to quality models of the ISO/IEC 9126, McCall, Boehm and so on. In figure 6.7 an overview of the hierarchical quality model for design patterns related to object oriented programming is given.

Figure 6.7.: Quality Model for Design Patterns related to Object oriented Programming [KG04]

6.4. Quality Evaluation Processes

In this chapter we gave an overview of general and specialized quality models, so far. However, several issues like the impact of a particular sub-characteristic - in terms of the ISO/IEC 9126 - on a main characteristics have not considered. Further, we did not go into detail about how the characteristics can be determined by use of metrics. In this section we bridge the gap between quality models and metrics by giving an overview of quality evaluation processes that are concerned with the validation of quality models and the relationship between characteristics and metrics.

6.4.1. Khosravi and Guéhéneuc

In [KG05] Khosravi and Guéhéneuc critically discuss open issues related to quality models and quality characteristics in general. The first issue addressed by Khos-
ravi and Guéhéneuc is related to the judgment of quality based on the humans perception, that is thought of as being the best source of measurement of quality as it is the person who needs to deal with the quality of software. However, humans usually have a different taste and value of quality. A feasible solution of this issue is to use a categorization of people that deal with the software. For instance, should software architects have a different taste of a software's quality compared to the user. However, the divergence between the taste of quality within a particular group is expected to be smaller than across groups.

A second issue addresses software metrics that are thought of as being the only mechanized tool for evaluating the internal attributes of software [KG05]. Unfortunately, the evaluation of software code is not enough as only a small number of attributes related to quality are addressable. Further, they identified that it is hard to find reasonable metrics for quality that is accepted by all people [KG05]. A possible solution for this issue proposed by Khosravi and Guéhéneuc is to modify the metrics during runtime for better results.

Finally, they discuss the issues related to quality models and the unspecified influence of sub-characteristics on the main-characteristics in hierarchical quality models. Usually, it is not further defined to which degree a particular sub-characteristic has an influence on a main characteristic. Khosravi and Guéhéneuc use the example of adaptability and installability that according to the ISO/IEC 9126 are sub-characteristics of portability. They raise the question of whether the influence of adaptability compared to installability is $\frac{2}{3}, \frac{3}{2}, \frac{1}{2}, \ldots$. Obviously, the solution to this issue is to come with coefficients addressing the influence of the sub-characteristics on the main characteristics for a particular group and domain.

Based on these issues and possible solutions Khosravi and Guéhéneuc introduce a method for defining a meaningful quality model and related metrics to assess the equality of software. The methodology is comprised of nine steps:

1. Choosing Category of People,
2. Identifying Sample Program (BP),
3. Building a Quality Model that consist of two steps:
   - Choosing a super-characteristic.
6.4. Quality Evaluation Processes

- Choosing and organizing characteristics related to the super-characteristic.

4. Human Evaluation,
5. Computing Software Metrics over BP,
6. Machine Learning Tools,
7. Computing Software Metrics over Example Programs,
8. Adapting Metric,

The methodology is applied to a case study to build and to apply the quality model considering program architectures [KG05]. The resulting quality model is presented in figure 6.8, were we refer to [KG05] for further details on the related metrics and the methodology in general.

![Quality Model for Software Reuse](image_url)

Figure 6.8.: Quality Model for Software Reuse [KG05]

6.4.2. ISO/IEC Quality Engineering

In 2001 the ISO/IEC 14598 was introduced as a standard that provides an evaluation process that can be applied to the generic measurement process defined in the ISO/IEC 9126. Although the ISO/IEC 14598 was introduced prior to the ISO/IEC
6. Quality Model - Related Work and Evaluation Processes

9126 it has not been further regarded in its revised version. The *Software Product Quality Requirements and Evaluation (SQuaRE)* project aims to provide a second generation of quality standards including the harmonization of the terminology used in different standards.

Although the ISO/IEC 14598 provides a generic linkage between the evaluation process and the quality model related metrics it does not yet specify a format of specific prescriptive quality engineering practices [SAA03]. The linkage between the two standards is illustrated in figure 6.9.

![Figure 6.9: ISO/IEC 14598 Evaluation process [SAA03]](image)

6.4.3. Moraga et al.

In contrast to the aforementioned methodologies, which have their the focus on providing a methodology to determine and validate a quality model and related metrics, in this section we focus on a particular facet of these methods that is discussed more in detail in [MBM+08]. Basically, in the ISO/IEC 9126 a good internal quality of a software product is thought of as being a prerequisite for good external quality that in turn is thought of as being a prerequisite of a good quality in use. In
[MBM⁺08], Moraga et al. changed the focus of assessing the quality of a software application towards quality in use as being the driving factor.

Based on the internal/external quality characteristics and sub-characteristics and the quality in use characteristics defined in ISO/IEC 9126, Moraga et al. applied a statistical method based on Bayesian Belief Networks (BBN) to identify the relationship between the in-use and external characteristics. Compared to the principle component analysis or linear regression that require initial numerical values to determine statistical relationships, BBN’s have the advantage that these are based on a directed acyclic graph with uncertain variables representing the nodes of the graph. The edges, instead, are the casual or influential links between the uncertain variables.

To determine the relationships between the external and in use characteristics, these are used as uncertain variables of a BBN. Further, the relations amongst these variables are determined, where Moraga et al. used a sub-set of the relations between external (sub-characteristics) and in use characteristics as being of significant value. In a second step the directed acyclic graph is built. Finally, in a third step the conditional probabilities (conditional probability tables are used) are associated with the nodes. The conditional probability tables introduced are based on the categorization of the ISO/IEC 1061 in acceptable, marginal and unacceptable, where they are used to determine the weight of the links. These are further used to calculate the probability distribution of each node.

The resulting BBN can be used to determine the influence of the external quality characteristics on the overall quality. However, Moraga et al. used a common practice for BBN where they simplified the relationships amongst the nodes by introducing synthetic nodes. Basically, they reorganized the directed graph, where the intermediate nodes correspond to the external (main) characteristics that affect a quality in use characteristic. In figure 6.10 the directed graphs addressing each quality in use characteristic is presented.

Moraga et al. clearly stated that the weights in the conditional tables of each node may need to be tailored to a particular domain and user group. However, the underlying method is still valid and can be applied to determine the influence of sub-characteristics on main characteristics (internal and external) and as proposed by Moraga et al. on quality in use characteristics.
6. Quality Model - Related Work and Evaluation Processes

Figure 6.10.: Relationship between the External and Quality In Use Characteristics [MBM+08]

6.5. Summary

In this chapter we gave an overview of traditional and general quality models being commonly accepted in the literature. Others, however, like the quality model of Firesmith [Fir03] and the research results from Dromey [Dro95] exist. The most commonly accepted but also the most discussed quality model is the ISO/IEC 9126 being currently under revision in the ISO/IEC SQuaRE project.

Beside the traditional and general models that need to be adapted to a particular point of view (user group) and domain it should be applied to, two specialized quality models for COTS components and component reuse were presented in this chapter. The latter is the result of the application of the methodology of Khosravi and Guéhéneuc. Other methods to identify a valuable component model and related metrics are discussed in the preliminary research results of the ISO/IEC SQuaRE project. Both, the methods introduced by Khosravi/Guéhéneuc and the SQuaRE project are intended to develop a quality model, to identify related metrics and to
validate these results based on a collection of expert opinions e.g. by use of questionnaires.

As the development of a quality model addressing a particular issue such as hierarchical component models and related metrics is a long term process we do not go into further detail in this issue. The different categories of the hierarchy-aware consideration have a major influence on the quality characteristics and related quality models. As we will show in chapter 9 HIERARCHY-AWARE MEASUREMENT for example the delegation ratios of the functionality encapsulated by the invocation chain of an operation has a major influence on the reliability of a composition hierarchy.
Chapter 7

Software Measurement

In the previous part we identified KobrA as having the most general component model compared to the underlying component models of the other frameworks discussed. We use KobrA’s terminology and perspectives for the development of the measurement process - parametrized measurement - and metrics for hierarchical component based system. The metrics, however, are based on the component model defined in chapter 5 HIERARCHICAL COMPONENT MODEL.

As well as being a model upon which software metrics are defined, theoretical foundations are a pre-requisite for the development of meaningful software metrics. Before we turn our attention to hierarchy-aware metrics in chapter 9 HIERARCHY-AWARE MEASUREMENT and an overview of well known software metrics and metrics suites in the next chapter, we first focus on theoretical foundations of software metrics in this chapter.

Unfortunately no clear consensus is found on how particular terms of measurement theory are interpreted in the field of software measurement. In order to set a solid foundation for a method to theoretically validate hierarchical software metrics, the basic terminology and principles of measurement theory necessary for the “validation method” are presented in section 7.2 MEASUREMENT THEORY.
Before we go into detail on the terminologies and principles of measurement theory, we give a brief overview of basic terms and definitions related to software measures and software metrics in the next section. A measurement theory based definition of basic terms - software metrics and software measures - is given in section 7.3 Measurement Theory based Definitions of Software Measures/Metrics. Basic properties of software measures presented in the recent literature are interpreted in terms of the principles of measurement theory in section 7.4 Properties of Software Measures. Based on the properties of software metrics a measurement theory based method for the theoretical validation of software metrics is introduced at the end of this chapter.

7.1. Basic Definitions of Software Metrics

In [GBC+06] initial steps towards a consistent terminology for software measurement is presented, finally resulting in the Software Measurement Ontology (SMO). Garcia et al. compared definitions of particular terms related to software measurement used in the literature and identified that often no clear consensus is found on these terms.

7.1.1. Software Measures vs Software Metrics

Even between standards of the same organization inconsistencies between the definition of particular terms are found, e.g. the definition of the term “metric” in IEEE Std. 610.12 [IEE90] is different from the one found in IEEE Std. 1061 [IEE98]. However, in the literature the terms software measure and software metric (metric, quality metric) are used synonymously by some authors as well. For instance, Briand et al. [BMB02] make use of a similar definition for the term measure, as the ISO/IEC Std. 14598 [ISO99] does for the term metric.

Other authors, instead, clearly distinguish between the terms. The “Groubstake group” makes use of the science of measurement theory for the foundations of software measurement. Being consistent with the interpretation of the terms metric and software measure of [CDS86], the “Groubstake group” presents a set of four criteria that need to be satisfied. The resulting mapping is then called a software
7.1. Basic Definitions of Software Metrics

measure [BBF‘90]. In contrast, a metric is defined by satisfying at least the fourth criteria. The set of criteria used as the basis for a distinction between the terms are presented next. Examples for the criteria are found in [BBF‘90].

To apply the “basic criteria of measurement theory to software measures requires the identification and/or definition of”

- attributes of software products and processes. These attributes need to be aspects of software that have both intuitive and well-understood meanings.
- formal models or abstractions which capture the attributes.
- important relationships and orderings which exist between the objects (being modeled) and which are determined by the attributes of the models.
- mappings from the models to number systems which preserve the order relationships.

This idea is the basis of the representational theory of measurement.

In the following we often use the terms metric and measure synonymously for the terms software metric and software measure for short.

In section 7.3 Measurement Theory Based Definitions of Software Measures/Metrics the “Groubstake group’s” definition of a software metric and a software measure is refined by applying the basic definitions of measurement theory to the definition of both terms.

7.1.2. Direct and Indirect Metric

Qualified versions of the basic terms, metric and measure, are found in the literature as well. The most commonly used distinction is between direct and indirect metrics. Based on the definition of the ISO/IEC Std. 14598 [ISO99] we define a direct metric/measure as:

Definition (Direct Metric). A direct metric is a metric that does not depend upon any other metric.

The term direct measure is defined analogously by replacing the terms metric with measure. However, in the literature other terms are used to describe a direct/indi-
7. Software Measurement

*rect metric* as well. For instance the term *fundamental metric* is used by Fenton [Fen91] as a synonym for the term *direct metric*.

Well known *measures* are the lines of code, the number of branch statements or the number of methods of a software entity. Assuming that these fulfill the criteria of a *software measure*, these can be thought as of being direct as they do not depend on any other *software measure*.

In contrast to a *direct metric* an *indirect metric* is defined as:

**Definition** (Indirect Metric). An indirect metric is defined as a metric that is derived from one or more other metrics.

Like the term *direct measure* an *indirect measure* is defined analogously. The definition of an *indirect measure* corresponds to the definition used in the ISO/IEC Std. 14598 [ISO99]. As for the term *direct metric*, other terms such as *derived metric* [Fen91] are used synonymously for the term *indirect measures/metrics*.

McCabe’s cyclomatic complexity [Mcc76] is thought of being an *indirect measure* as it is derived from the *measures* number of nodes and number of arcs in a control flow graph of a software entity.

7.1.3. Quality Based Software Metric

A different categorization of *software metrics* is derived from the field of *quality assessment*. Actually, the *quality* of a software entity can be measured at different stages of the development process. A categorization into *internal*, *external* and “*in use*” metric is used analogously to the categorization of the so called quality characteristics in the ISO/IEC Std. 9126 [ISO91].

Based on the definition used in the ISO/IEC Std. 8402 (Quality vocabulary) [ISO86] derived from N. Bevans investigation of “quality in use metrics” [Bev99], *quality* is defined as:

**Definition** (Quality). *Quality describes the totality of characteristics of an entity that bear on its ability to satisfy stated and implied needs.*

**Definition** (Quality Characteristics). *Quality characteristics, in turn, can be thought of being a feature or characteristic affecting an item’s quality [IEE90], such as reliability, maintainability and usability.*
7.1. Basic Definitions of Software Metrics

The ISO/IEC Std. 9126 [ISO91] and several other publications present software metrics to “measure” quality characteristics. According to [BTV06] these metrics should be thought of more as being indicators for the quality characteristics rather than as being interpreted as being metrics of these. Based on the definition of the term indicator in [ISO99] we define an indicator metric/measure as:

**Definition** (Indicator Metric). *An indicator metric is defined as giving an indication of a particular quality characteristic.*

Like direct and indirect measures, an indicator measure is defined analogously to an indicator metric. Based on the categorization of quality characteristics in [ISO91] internal, external and in use indicator metrics are defined as:

**Definition** (Internal Indicator Metric). *An internal indicator metric is defined as a metric related to a static representation of a software entity.*

*Internal indicator metrics* can be obtained for instance by inspection of a source code representation of a software entity. In contrast to the *internal indicator metrics* the *external indicator metrics* are defined as:

**Definition** (External Indicator Metric). *External indicator metrics are related to a software entity that is executed under stated conditions.*

The difference between *external indicator metrics* and the *in use indicator metrics* is the context in which these are executed. *External metrics* can be applied early in the development process for instance during a unit test of a software entity. In use indicator metrics, instead, represent metrics that are executed “in specified contexts of use” that involves user interaction as well. These are defined as:

**Definition** (In Use Indicator Metric). *In Use Indicator Metrics are defined as metrics that are obtained during execution of a software entity under stated conditions in specified contexts of use.*

Like the categorization of direct and indirect metric, internal, external and in use indicator measures are defined analogously to the corresponding metrics. Being aware of the fact that these *metric* only being *indicators of quality characteristics* rather than *metric* of these, we use the terms internal, external and in use metric for short.
7. Software Measurement

7.2. Measurement Theory

The relevance of measurement theory for software measurement has often been discussed in the recent years since it was first introduced in the early 1990’s.

As mentioned earlier, the “Grubstake group” was the first to apply measurement theory to software measurement. In [BBF+90], Baker et al. present basic properties to distinguish between the terms measures and metrics (see also section 7.1 Basic Definitions of Software Metrics), discuss the problems of lack of validation of software metrics, and they introduce the first ideas for a methodology to define structured metrics.

Foundations of measurement theory, like extensive measurement, have been applied to the field of software measurement. Bollmann-Sdorra and Zuse [ZB89] were the first that applied extensive measurement to software measurement: more precisely they applied the theory of extensive measurement to the measurement of complexity metrics. More generally Bollmann-Sdorra and Zuse applied the measurement theory presented by Roberts [Rob79] and Kantz et al. [KLST71] on the foundations of software measurement. However, particular terms used by Roberts and Kantz et al. have been interpreted and adapted to the field of software measurement. Some of their basic assumptions such as the extensive measurement being of major importance for complexity measures and the interpretation of particular terms has lead to controversially discussions between Konrad and Bollmann-Sdorra/Zuse in the subsequent years. In several publications the different positions of both parties have emotionally been discussed by Konrad [Kon91, Kon92] and Bollmann-Sdorra/Zuse [ZB89, ZBS92] in what Wexelblat called the “Battle of Berlin” [Wex92]. Despite the criticism of Konrad, the foundations of software measurement based on measurement theory presented by Bollmann-Sdorra and Zuse widely gained acceptance in the past recent years.

However, for the development of software metrics for hierarchical component-based systems/methodologies properties like extensive measurement loom large as we will see in chapter 7.4 Properties of Software Measures. Therefore, we turn our attention to the foundations of measurement theory and give a brief overview of some basic definitions and notions of measurement theory used by Roberts [Rob79].
7.2. Measurement Theory

7.2.1. Foundations of Measurement Theory

Measurement theory is concerned with assigning numbers to objects and phenomena of the real world, to describe the relation between objects such as “is heavier than”, “is equal to”, and so on. Further, it is concerned with describing relations between objects after being affected by binary operations, such as relating the combined mass of two objects \( a \) and \( b \) with the mass of object \( c \). Similar to the introduction to the field of measurement theory by Roberts [Rob79] we make use of real world examples like mass and temperature to allow a better understanding of the terminology and definitions introduced.

7.2.1.1. Notations and Terminologies

In [Kon91] Konrad identified that Bollmann-Sdorra and Zuse in [ZB89] interpret the definitions and notions of measurement theory presented by Roberts [Rob79] in a way, which from Konrad’s perspective lead to a misuse of these (see also section 7.4.1.4 Viewpoints and Sub-Concepts). In the following we give an overview of the basic definitions and notions used, for instance, to define extensive measurement. We start by defining relations and introduce the basic properties of relations and relational systems afterwards.

Relations

The Cartesian product \( A \times B \) between the objects \( (a_1, a_2, \ldots, a_n) \) of set \( A \) and the objects \( (b_1, b_2, \ldots, b_m) \) of set \( B \) is described by the ordered pair \( (a, b) \) with \( a \in A \) and \( b \in B \). More generally, the Cartesian product between the sets \( A_1, A_2, \ldots, A_n \) \( (A_1 \times A_2 \times \cdots \times A_n) \) is defined by the ordered n-tuple \( (a_1, a_2, \ldots, a_n) \).

A simple example of the Cartesian product is given by comparing the observed mass of the entities of a small apple crate (representing set \( A \)) which consists of 4 pieces. By lifting the four apples \( \{a_1, a_2, a_3, a_4\} \in A \) we may have observed that \( a_4 \) is the heaviest. Apple \( a_3 \) has a higher mass than \( a_2 \), which in turn is observed to have a higher mass than \( a_1 \). The Cartesian product \( R = A \times A \)
representing the pairwise comparison of masses of $a \in A$ is then given by:

$$R = (a_i, a_j), \text{ with } i, j = 1, 2, 3, 4 \text{ and } a \in A.$$ 

A binary relation is defined as a subset of the Cartesian product. For the given example the relation $R'$ between the observed apples' masses related to “is higher than” results in:

$$R' = \{(a_2, a_1), (a_3, a_2), (a_3, a_1), (a_4, a_3), (a_4, a_2), (a_4, a_1)\}.$$ 

More formally, an ordered pair $(a, b)$ with $a, b \in A$ is only a binary relation $R'$ if $a > b$. Referring to Roberts [Rob79] we write $aRb$ for short if $(a, b)$ is a binary relation $R$.

It should be noticed that the properties of a relation in general are only defined with its underlying set. Roberts makes use of the “brother of” relation with the underlying sets $A$ representing all the people of the United States and the subset $B$ representing only the males in the United States to show that the relation has different properties depending on the restrictions caused by the underlying sets. For example, some kind of symmetry properties are found if the underlying set is $B$, as if $a$ is “brother of” $b$ then $b$ is “brother of” $a$ as well ($(a, b) \in R$ then $(b, a)$ is in $R$). This symmetry, however, is not found, if $A$ is used as the underlying set. As $A$ contains females as well, the relation between $a$ being a male and “brother of” $b$ being a female would be in $R$, however $bRa$ would obviously not. As it can be seen by this simple example the properties of relations between objects may be affected by the underlying set, so that in the following we use the term relational systems defined by $(A, R)$ rather than relation according to [Rob79].

**Properties of Relations**

For the introductory example of the relation system “brother of” for the set of males in the United States a basic property of relational systems, the symmetry, has already been mentioned. More formally, symmetry is defined as the binary relation $(A, R)$ being symmetric if for all $a, b \in A, aRb \Rightarrow bRa$. However, other properties related to symmetry are found as well. For instance, the “greater than” relation for real numbers. If a number $a$ is “greater than” $b$, with $a, b \in \mathbb{R}$ than $b$ is not “greater than” $a$. The symmetry property related to the relation “greater than” on $\mathbb{R}$ is called asymmetric. A third symmetry related property is the so called antisymmetry which
is found, for instance, in the “greater than or equals” relation of real numbers. Finally, relations not having any symmetry property are called *nonsymmetric*. The formal definitions of the symmetry properties are given in the following table (Table 7.1):

**Symmetry:** A binary relation \((A, R)\) is *symmetric* if for all \(a, b \in A\), \(aRb \Rightarrow bRa\)

**Asymmetry:** A binary relation \((A, R)\) is *asymmetric* if for all \(a, b \in A\), \(aRb \Rightarrow \neg bRa\)

**Antisymmetry:** A binary relation \((A, R)\) is *antisymmetric* if for all \(a, b \in A\), \(aRb \text{ and } bRa \Rightarrow a = b\)

**Nonsymmetry:** A binary relation \((A, R)\) being *nonsymmetric* means that a relation is not symmetric

Table 7.1.: Symmetry properties of relational systems

A different kind of property is found for the “father of” relational system \((A, R)\) with the underlying set \(A\) representing the people on the world. A father can obviously not be a father of himself. This kind of property is called *irreflexive*. However, the “is higher than or equals” relation \(R'\) for the observed mass of apples \(A\) is different, as the binary relation \(aRa\) is true for all \(a \in A\). Then the binary relation is defined to be *reflexive*. The formal definitions of the reflexivity properties are given in the following table (Table 7.2):

**Reflexivity:** A binary relation \((A, R)\) is reflexive if for all \(a \in A\), \(aRa\)

**Irreflexivity:** A binary relation \((A, R)\) is irreflexive if for all \(a \in A\): \(\neg aRa\)

**Nonreflexivity:** A binary relation being nonreflexive means that a relation is not reflexive.

Table 7.2.: Reflexivity properties of relational systems
The introductory example, the “is higher than” relation of the observed apples’ mass, has another fundamental property. Knowing that the observed mass of apple $a$ “is higher than” the observed mass of apple $b$ and knowing that the observed mass of apple $b$ “is higher than” the observed mass of apple $c$ results in the observed mass of $a$ being “higher than” $c$ as well. This property is called \textit{transitivity}. The binary relational system $(A, R)$ “is greater than” on real numbers is an example of a relational systems being \textit{negatively transitive}. A binary relation $(A, R)$ is called \textit{negatively transitive} if, for all $a, b, c \in A \neg aRb \& \neg bRc \Rightarrow \neg aRc$. In other words, for the relation $R$ “is greater than” on the real numbers $A$, this means that a number $a$ not being greater than $b$ is “smaller than” $b$. The same is true for a number $b$ not being greater - therefore being smaller - than $c$. Since $a$ is “smaller than” $b$ and $b$ is “smaller than” $c$ this implies that $a$ is also “smaller than” $c$ - in other words $a$ is not “greater than” $c$ - and therefore being a \textit{negatively transitive} relational system. A formal definition of both properties, \textit{transitivity} and \textit{negatively transitivity}, is given in Table 7.3.

\begin{table}[h]
\begin{tabular}{|c|c|}
\hline
\textbf{Transitivity} & A relation $(A, R)$ is called \textit{transitive} if, for all $a, b, c \in A$, whenever $aRb$ and $bRc$, it implies that $aRc$. In symbols, $(A, R)$ is transitive if $aRb \& bRc \Rightarrow aRc$ \\
\hline
\textbf{Negatively Transitivity} & A binary relation $(A, R)$ is called \textit{negatively transitive} if, for all $a, b, c \in A$, $\neg aRb \& \neg bRc \Rightarrow \neg aRc$ \\
\hline
\end{tabular}
\caption{Transitivity properties of relational systems}
\end{table}

\section*{Equivalence Relations}

As Roberts [Rob79] notes most binary relations satisfy the three properties \textit{reflexivity}, \textit{symmetry} and \textit{transitivity}. Binary relations satisfying these three properties are called \textit{equivalence relations}.

Let $(A, R)$ be an equivalence relation and $a \in A$, with $a^* \text{ denoting } \{b \in A : aRb\}$. The set $a^*$ is then called an \textit{equivalence class} and an element $a$ is called a representative of the equivalence class $a^*$. For example the “modulo 2” relation on the natural num-
bers \( \mathbb{N} \) results in two equivalence classes 1* and 2*, with 
1* = 0, 2, 4, 6, ... and 2* = 1, 3, 5, 7, ....

The collection of equivalence relations \((A, R)\) partitions \(A\), which means that every object of \(A\) is in a particular equivalence class. Two equivalence classes can be categorized as either being identical or as being disjoint (Theorem 1.1 in [Rob79, p. 26]). A major advantage of dealing with equivalence classes is that it is sufficient to deal with a representative of the equivalence class rather than dealing with each object of \(A\).

**Order Relations**

In this section we turn our attention on order relations based on the introduced definitions. Beside the binary relation \(R\) “is heavier than” related to the observed mass of apples \(A\), other binary relations can easily be applied to \(A\) as well. An alternative binary relation \(P\) on \(A\) is related to preferring an object \(a \in A\) to \(b \in A\). The binary relation \(W\) “is weakly preferred to” \(b\), for example, means that we either prefer \(a\) to \(b\) or are indifferent between \(a\) and \(b\). Denoting the binary relation of being indifferent between \(a\) and \(b\) as \(I\), we get

\[ aWb \iff (aPb \text{ or } aIb). \]

Additionally, of being reflexive and transitive the relational system \((A, W)\) has the property of being strongly complete. Strongly complete means that for every \(a\) and \(b \in A\) (including \(a = b\)) either \(aWb\) or \(bWa\). If in contrast, for \(a\) and \(b \in A\) either \(aWb\) or \(bWa\) is true except for \(a = b\) then \(W\) is meant to be complete.

Based on the binary relations’ properties defined earlier and the properties strongly complete and complete introduced in this section, binary relations can be categorized as being in a particular order. A binary relation fulfilling at least the properties of being transitive, antisymmetric and strongly complete is characterized as being a simple order. An overview of order types is given in table 7.4.

**Functions and Operations**

Up to this point objects and relations between objects of a set \(A\) have been considered in an isolated way. However, typically operations are performed on the objects
Table 7.4.: Order Relations based on [Rob79, Table 1.3]

<table>
<thead>
<tr>
<th>Property</th>
<th>Quasi Order</th>
<th>Weak Order</th>
<th>Strict Weak Order</th>
<th>Simple Order</th>
<th>Strict Simple Order</th>
<th>Partial Order</th>
<th>Strict Partial Order</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reflexivity</td>
<td>×</td>
<td>×</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Symmetric</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Asymmetric</td>
<td>×</td>
<td>×</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Antisymmetric</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Transitive</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td></td>
</tr>
<tr>
<td>Negatively Transitive</td>
<td>×</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Strongly complete</td>
<td>×</td>
<td>×</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Complete</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>×</td>
</tr>
</tbody>
</table>

as well. Taking the apple crate consisting of 4 apples from the beginning of this section, we have relations between the mass of each individual object of $A$. Often a combined consideration of the objects is performed as we will see in detail in the next section.

Generally, operations $f$ performed on the objects of $A$ ($f : A \rightarrow A$) fulfilling the properties

$$(\forall a \in A) (\exists b \in A) (aRb)$$

$$(\forall a, b, c \in A) (aRb \& aRc \Rightarrow b = c)$$

are called binary operations. Examples of binary operations are the “$+$”-operation in the context of adding the masses of apples to perform a combined comparison between objects. A different example of the “$+$”-operation is given for $\mathbb{R}$ as underlying set, resulting in the binary operation $(\mathbb{R}, \circ)$. An operation $\circ$ defined as

$$\circ(a, b, c) \Leftrightarrow c = a + b$$

is a binary operation. However, other relations such as $(A, \circ)$, with

$$\circ(a, b, c) \Leftrightarrow c = a/b,$$

are not operations as, for instance, the relation $\circ(1, 0, c)$ is not defined in $\mathbb{R}$.

**Relational Systems**

Although binary operations performed on a set $A$ can be treated as binary relations, as mentioned in the previous section, these are typically singled out. A $n + m + 1$-
7.2. Measurement Theory

tuple \( U = (A, R_1, R_2, \ldots, R_n, \circ_1, \circ_2, \ldots, \circ_m) \) is used to describe both the (not necessarily binary) relations \( R_1, R_2, \ldots, R_n \) and the binary operations \( \circ_1, \circ_2, \ldots, \circ_m \). The \( n + m + 1 \)-tuple \( U \) is called a relational system.

7.2.2. Fundamental and Derived Measurement

In the previous section we introduced notions and definitions on relations and operations between objects of a predefined set \( A \), for instance the mass of apples. The introduced notions and definitions have been described for objects of the real world, without interpreting them e.g. in a numerical way. The process of assigning numerical values to objects is called the measurement process. Two different kinds of measurement are defined in [Rob79]: fundamental and derived measurement.

Fundamental measurement deals with the measurement process of what Roberts [Rob79] calls fundamental measures. As presented previously, the domain of software measurement makes use of the term direct measures to represent the term fundamental measures. The second measurement process - the derived measurement - deals with derived measures - in software measurement mainly the term indirect measures is used - which depend on other measures, for instance, by being a combination of direct measures.

Taking the apple crate \( A \) consisting of 4 apples \( a_1, a_2, a_3, a_4 \). From the introduction of this section we have already described the observations related to the relations between the observed masses of the elements of \( A \). After assigning numerical values \( f(a) \) to the objects these need to satisfy the properties derived from the objects. For instance, after assigning numerical values \( f(a) \) to the mass of objects \( a, b \in A \), the “is heavier than” relation \( H \) should lead to \( aHb \iff f(a) > f(b) \). For special binary operations, however, the application of numerical values on combined objects is equal to the combined application of the individual objects. In case of the binary operation “+” performed on the numerical interpretations of the apples’ mass \( (a_1 \) and \( a_2 \), the mass resulting from the measurement of the individual objects and the application of the combination operations is equal to the mass of a combined “object” \( (a_c = a_1 \land a_2) \).

More formally, this can be described as,

\[ aRb \iff f(a) > f(b) \quad (7.1) \]
which also preserves the binary operation ◦ so that for all \( a, b \in A \):

\[
f(a \circ b) \Leftrightarrow f(a) + f(b)
\]  

(7.2)

### 7.2.2.1. Homomorphisms of Relational Systems

The goal of mapping empirical observations to numerical interpretations can be thought as of being a primary goal of measurement. The mapping \( f \) of the empirical relational system \( U = (A, H, \circ) \) related to the observed mass of apples, for instance, to a numerical relational system \( \mathcal{B} = (\mathbb{R}, >, +) \), should preserve the relations and operations of \( U \). Relational systems such as \( U = (\mathbb{R}, >, +) \) are called numerical relational systems if the underlying set is a set of numbers, e.g. real numbers. However, sometimes the term formal relational system is used synonymously for numerical relational system in the literature e.g. [BEM96].

Roberts presents an abstraction of relational systems by its type. The type of a relational system \( U = (A, R_1, R_2, \ldots, R_n, o_1, o_2, \ldots, o_m) \) is defined on the basis of its relations and operations. The type of \( U \) is defined as \( (r_1, r_2, \ldots, r_n; m) \), with \( r_i \) being \( p \) for \( p \)-ary relations. Therefore, the relational system \( U = (A, H, \circ) \) has the type \((2;1)\).

Relation- and operation-preserving mappings from one relational system \( U \) to another relational system \( \mathcal{B} \) are called homomorphisms. It should be noted that a mapping \( f \) does not necessarily need to be a one-to-one or onto relation. This special kind of homomorphism, where a one-to-one or onto mapping is preserved, is called an isomorphism. Formally, a homomorphism is defined as:

**Definition 1** (Homomorphism). A mapping \( f \) from one relational system \( U \) to another relational system \( \mathcal{B} \) preserving all relations and operations is called a homomorphism. The relational system \( \mathcal{B} = (B, R'_1, R'_2, \ldots, R'_p, o'_1, o'_2, \ldots, o'_q) \) has the same type as \( U \). Then a function \( f : A \to B \) is called a homomorphism from \( U \) into \( \mathcal{B} \). For all \( a_1, a_2, \ldots, a_r \in A \),

\[
R_i(a_1, a_2, \ldots, a_r) \Leftrightarrow R'_i[f(a_1), f(a_2), \ldots, f(a_r)], i = 1, 2, \ldots, p.
\]  

(7.3)

and for all \( a, b \in A \),

\[
f(a \circ b) = f(a) \circ'_i f(b), i = 1, 2, \ldots, q
\]  

(7.4)
If it is possible to assign a homomorphism from an empirical relational system \( U \) to (usually) a numerical relational system \( B \), the measurement process is called fundamental measurement.

### 7.2.2.2. Representation and Uniqueness Problem

Two fundamental problems need to be discussed before a measurement process can be performed. The first basic problem is called the representation problem. The representation problem is related to discovering necessary and sufficient conditions for the existence of an homomorphism from an empirical relational system \( U \) to a numerical relational system \( B \). Generally the emphasis should be put on discovering sufficient conditions, as all the sufficient conditions are necessary as well. Beside discovering the necessary and sufficient conditions for an homomorphism, empirically verifiable criteria for the homomorphism need to be found as well. Generally, the term axiom is used to describe the representation, where the term representation theorem is used to describe the theorem stating the sufficiency.

The uniqueness problem is the second basic problem in measurement theory. The uniqueness problem is related to the question of how unique a homomorphism is. The uniqueness problem is of major importance as it tells us which kind of scale a mapping \( f \) from an empirical relational system to a numerical relational system has. The latter gives rise to a theory of meaningfulness of statements involving scales [Rob79]. Furthermore, the uniqueness theorem puts limitations on the mathematical manipulations that can be performed on the numbers arising from the underlying scale values.

### 7.2.2.3. Regular Scales

A scale is defined as the triple \((U, \mathcal{B}, f)\). Both \( U \) and \( \mathcal{B} \) are relational systems and \( f \) is a homomorphism from \( U \) into \( \mathcal{B} \). In case of \( \mathcal{B} \) being a numerical relational system, the scale is called a numerical scale. A categorization of scale types can be performed on the basis of the uniqueness and meaningfulness of \( f \). A statement involving numerical scales is defined to be meaningful if the truth (or falsity) of the statement remains unchanged if a scale \((U, \mathcal{B}, f)\) is replaced by another scale \((U, \mathcal{B}, g)\).
7. Software Measurement

The *admissible transformations* applicable on a scale \((\mathcal{U}, \mathcal{B}, f)\) can be used to determine its meaningfulness. With \(A\) being the underlying set of \(\mathcal{U}\), \(B\) being the underlying set of \(\mathcal{B}\) and \(f\) being a homomorphism from \(\mathcal{U}\) into \(\mathcal{B}\) we define a function \(\phi\)

\[
\phi(A) = f(a) : a \in A
\]

that maps the range of \(f\) into the set of \(B\). Then the composition \(\phi \circ f\) is a function from \(A\) into \(B\). We say that \(\phi\) is an *admissible transformation of scale* if it is a homomorphism from \(\mathcal{U}\) into \(\mathcal{B}\).

Examples of admissible transformations [Rob79]: Let \(\mathcal{U} = (\mathbb{R}, \succ)\) and \(\mathcal{B} = (\mathbb{R}, \succ)\) be relational systems. Let \(f\) be a homomorphism from the underlying set of \(\mathcal{U}\) to the underlying set of \(\mathcal{B}\) (\(f : \mathbb{N} \to \mathbb{R}\)) given by \(f(x) = 2x\). A second homomorphism is given by \(\phi(x) = x + 5\) resulting in a composed function \((\phi \circ f)(x) = 2x + 5\), with \(x > y\), if and only if \(2x + 5 > 2y + 5\).

Thus \(\circ : f(A) \to B\) is an admissible transformation of scale.

**Definition 2** (Regular Scales [Rob79]). A scale \((\mathcal{U}, \mathcal{B}, f)\) is called regular, if there is a scale \((\mathcal{U}, \mathcal{B}, g)\) with a transformation \(\phi : f(A) \to B\) such that \(g = \phi \circ f\). A representation \(\mathcal{U} \to \mathcal{B}\) is regular if, given two scales \(f\) and \(g\), we can map each into the other by an admissible transformation.

**Definition 3** (Meaningfulness for Regular Scales [Rob79]). A statement involving (numerical) scales is meaningful if and only if its truth or falsity is unchanged under admissible transformations of all the scales in question.

Roberts presents a characterization of regular scales with \((\mathcal{U}, \mathcal{B}, f)\) being a regular scale if and only if for every other homomorphism \(g\) from \((\mathcal{U} \to \mathcal{B})\), and for all \(a, b\) in \(A\), \(f(a) = f(b)\) implies that \(g(a) = g(b)\) [RF76]. Based on the characterization of regular scales in [Rob79, p. 60, Theorem 2.1] Roberts concludes that every isomorphism is regular.

7.2.2.4. Scale Type

The identification of the underlying scale type is a pre-requisite for most statistical methods typically applied in the empirical evaluation. Therefore, we go into detail on the different scale types in the following and give an overview of the applicable statistical methods at the end of this section.
If a representation $\mathcal{U} \to \mathcal{B}$ is regular then the class of admissible transformation defines how unique each scale is. Furthermore, the class of admissible transformations can be used to define a scale type. The scale types presented next are ordered from the strongest to the weakest.

**Absolute Scale**

Only a single admissible transformations is defined for the absolute scale, that is $\phi(x) = x$. This means that in practice the only way of measuring things is by counting them.

**Ratio scale**

Related to the admissible transformations of a scale the class of similarity transformations $\phi : f(A) \to B$, with

$$\phi(x) = \alpha x, \alpha > 0$$

defines a ratio scale.

An example of a ratio scales is the mass of objects. It is possible to identify a zero point and then change the unit of mass by multiplying a positive constant. Other examples of a ratio scale are the temperature measured in Kelvin.

**Interval scale**

In contrast to the temperature measured in Kelvin being a ratio scale, the temperature measured in the Fahrenheit or Celsius scale are not. This is because, if we want to apply a transformation from Celsius to Fahrenheit we need to vary both the zero point (changing $\beta$) and the unit (changing $\alpha$). More generally, the class of admissible transformations applicable on a scale defining an interval scale is given by all the functions $\phi : f(A) \to B$ of the form

$$\phi(x) = \alpha x + \beta, \alpha > 0.$$ 

These kind of functions are called positive linear transformation and the corresponding scale is called an interval scale.
7. Software Measurement

**Ordinal Scale**

The different ways of grading students as applied in different countries are examples of *ordinal scales*. In Germany, typically the scale from 1 to 6 is used to grade students, where the best grade is 1. In contrast, in the United States the grading of students is based on the scale reaching from A to E/F, with A being the best grade and E/F denoting unsatisfactory/failed. In Italy, however, the grading reaches from 1 (weakest) to 10 as the best grade. From an abstract point of view grading is unique only up to order. If a scale is unique only up to order, the admissible transformation are *monotone increasing functions* $\phi(x)$, that is, functions $\phi : f(A) \to B$ satisfying the condition

$$x \geq y \Leftrightarrow \phi(x) \geq \phi(y),$$

or equivalently the condition

$$x > y \Leftrightarrow \phi(x) > \phi(y).$$

**Nominal Scales**

In practice we often find transformations to a numerical scale, for example, made to determine a particular object in a set of objects. For instance, the numbers on the jerseys of football players are used to determine a particular individual. The actual number has almost no significance and a change to any other number would contain the same kind of information, which is the identification of an element of the underlying set $A$. Therefore, the class of admissible transformations defining the *nominal scale* is determined by the “all-to-one” functions $\phi$.

**Applicable Statistical Methods**

In the literature a large number of statistical methods are defined. Generally, these are only applicable for particular scale types. For instance, the mean of the numerical representation of the objects of a set having a nominal scale are almost without any meaning. Take the numbers on the jerseys of football players, for instance. It is obvious that the mean of the numbers does not have any meaning. In contrast, applying the mean on an ordinal scale, the grades of the students of a particular course, for example, the mean becomes meaningful. Based on [Lea05] we give an
7.2. Measurement Theory

Overview of the applicable statistical methods related to a particular scale type in the following tables (Table 7.5 and 7.6).

<table>
<thead>
<tr>
<th></th>
<th>Nominal scale</th>
<th>Ordinal scale</th>
<th>Interval scale</th>
<th>Ratio scale</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mode</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>Median</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>Range Statistics</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>Variance</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>Standard Deviation</td>
<td></td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
</tbody>
</table>

Table 7.5.: Appropriate descriptive statistical methods (adapted from [Lea05])

<table>
<thead>
<tr>
<th></th>
<th>Nominal scale</th>
<th>Ordinal scale</th>
<th>Interval scale</th>
<th>Ratio scale</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-Parametric</td>
<td>×</td>
<td>×</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Parametric</td>
<td></td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>Chi-Square</td>
<td>×</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mann-Whitney U</td>
<td>×</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Kruskal Wallis H</td>
<td>×</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Friedman analysis of variation ANOVA</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td></td>
</tr>
<tr>
<td>t-test</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>Spearman Correlation</td>
<td>×</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pearson Correlation</td>
<td></td>
<td>×</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 7.6.: Appropriate inferential statistical methods (adapted from [Lea05])

7.2.2.5. Derived Measurement

Up to this point we focused on fundamental measurement. However, a second kind of measurement called derived measurement is of major importance as well. Pfan-
zagl [Pfa71] stated that derived measurement is no measurement at all, as any property that is going to be measured by a numerical scale having an empirical meaning would also have a fundamental scale. Thus, the relation between the fundamental scales used to determine the derived scale would become an empirical law between the fundamental scales.

In other words, it should be possible to obtain the same scale either as a fundamental scale or a derived scale. However, in practice it is not always that easy to obtain a fundamental scale from empirical observations. For instance, Weyuker [Wey88] presented a set of nine properties a measure related to the complexity of software entities should have. However, it is still unclear how to measure complexity in practice. In practice several measures giving an indication of complexity are used, e.g. the cyclomatic complexity presented by McCabe [Mcc76] or the Halstead effort [Hal77] (Weyuker discusses in detail whether these measures fulfill the properties of complexity and can be thought of as derived measures for complexity of software entities). In physics derived measurement is widely used as well. The speed \( v \), for instances, depends on the time \( t \) an object needs to cover the distance \( s \).

More formally the following definition is used to determine derived scales:

**Definition 4 (Derived Scale).** Suppose \( u : A \to \mathbb{R} \) is an ordinal utility function, that is, a function satisfying \( aPb \iff u(a) > u(b) \).

Let \( v \) be a function on \( A \) with the property that \( u(a) > u(b) \) if and only if \( v(a) < v(b) \). Then \( v \) is a derived scale.

**Representation Problem**

As stated in section 7.2.2.2 Representation and Uniqueness Problem the fundamental problems - the representation and uniqueness problems - of a particular measurement process need to be solved for derived measurement processes as well. Roberts notes that the *representation theorem* applied to derived measurement will state sufficient conditions for the existence of a function satisfying the definition. In other words, a derived scale \( g \) and a set of real valued functions (primitive scales) \( f_1, f_2, \ldots, f_n \) on \( A \), satisfy a certain condition \( C(f_1, f_2, \ldots, f_n, g) \). Any function \( g \) satisfying the condition \( C \) will be acceptable. More concrete, the condition
C can be an equation relating $g$ to $f_1, f_2, \ldots, f_n$. For instance, the speed $C$ is defined as $C(s, t, v)$ and the equation relating $v$ to $s$ and $t$ is given by $v = s/t$.

**Uniqueness Problem**

For the analysis of the scale types in a derived measurement the *uniqueness problem* is of more importance. Two different senses of the term “uniqueness” are found. These depend on whether we allow the real valued functions $f_1, f_2, \ldots, f_n$ to vary.

In the case of speed we know that the underlying primitive scales can vary. For instance, the distance can be measured in $cm, m, km, \ldots$, the time in $s, min, h, d, \ldots$. However, both primitive scales are ratio scales. Usually we allow the scales $d$ and $t$ to vary. With $s'$ being the distance and $t'$ being the time in another allowable scale, then there are positive numbers $\alpha$ and $\beta$ such that $s'(a) = \alpha s(a)$ and $t'(a) = \beta t(a)$, for all $a$ in $A$. The derived scale of speed is given by:

$$v'(a) = \frac{s'(a)}{t'(a)} = \frac{\alpha s(a)}{\beta t(a)} = \frac{\alpha}{\beta} v(a).$$

Thus the transformation from $v$ into $v'$ can be accomplished by a similarity transformation.

If it is not allowed to vary the primitive scales $s$ and $t$, then the speed $v$ is defined uniquely in terms of $s$ and $t$.

**Admissible Transformations**

For derived measurement we distinguish between admissible transformations in the *narrow* and in the *wide sense*, when a derived scale $g$ is defined from primitive scales $f_1, f_2, \ldots, f_n$ by the condition $C$. We define a function $\phi : g(A) \rightarrow \mathbb{R}$ to be *admissible in the narrow sense* if $g' = \phi \circ g$ satisfies $C(f_1, f_2, \ldots, f_n, g')$.

In contrast, a function $\phi$ is defined as being *admissible in the wide sense* if there are acceptable replacement scales $f'_1, f'_2, \ldots, f'_n$ for $f_1, f_2, \ldots, f_n$ so that $C(f'_1, f'_2, \ldots, f'_n, g')$.

In the previous example of speed $v$ being derived from $s$ and $t$, the identity is the only admissible transformation in the narrow sense. However, we have identified that every admissible transformation in the wide sense - this is then the primitive scales of speed can vary - is also a similarity transformation.
Furthermore, we define a derived scale $g$ to be regular in the narrow sense, if whenever $C(f_1, f_2, \ldots, f_n, g')$ holds, there exists a transformation $\phi$ with $\phi : g(A) \to \mathbb{R}$ such that $g' = \phi \circ g$. Then the transformation $\phi$ is called an admissible transformation in the narrow sense [Rob79]. In contrast, if we have acceptable replacement scales for $f_1, f_2, \ldots, f_n$ and whenever $C(f'_1, f'_2, \ldots, f'_n, g')$ holds, there exists a transformation $\phi$ with $\phi : g(A) \to \mathbb{R}$ such that $g' : \phi \circ g$. In this case $\phi$ is called an admissible transformation in the wide sense.

Analogously to the definition of scales in the fundamental measurement, scale types are defined for the derived measurement. Like the definition of scales that are distinguished in scales in the narrow and wide sense, the same distinction is made for scale types. For example, to obtain a ratio scale in the narrow sense the class of admissible transformations in the narrow sense needs exactly to be the class of similarity transformations as Roberts shows in [Rob79].

### 7.2.3. Extensive Measurement

In [ZB89] Bollmann-Sdorra and Zuse applied the notions and terminologies of measurement theory to the measurement of software complexity measures. From their point of view they defined additivity to be a basic property a complexity measure for software entities should posses. The need for a relational system to be additive is often found for observation of real world objects such as the mass of apples’ that has been used to provide an introductory example of the notions and terminologies of measurement theory. In case of apples’ masses the need for a relational system to be additive is naturally grounded, as we want a combined object - the observed mass of two or more apples - to preserve the properties for individual objects. Traditionally, relational systems that are additive have been called extensive in measurement theory. The problem of finding conditions on $(A, R, \circ)$ being sufficient for the existence of a homomorphic map into $(\mathbb{R}, >, +)$ is called the problem of extensive measurement.

As mentioned earlier some “misleading” interpretations of definitions of measurement theory when applied to the measurement process of software entities have lead to the controversial discussion between Bollmann-Sdorra/Zuse and Konrad in the 1990’s. Bollmann-Sdorra and Zuse used the proof for a software complexity
measure having an extensive structure to identify if a particular measure has a ratio scale, since an extensive structure implies a ratio scale as presented in [Rob79]. Amongst other issues addressed by Konrad [Kon91], software measures for complexity having a ratio scale do not necessarily need to be additive, as only the conclusion that measures being additive implies that these have a ratio scale [Rob79]. Preferably software measures addressing the hierarchical refinement dependencies should be additive. Software measures being additive imply that these are recursively applicable on the components in the composition hierarchy in order to measure a particular facet of the underlying abstraction path. These are discussed more in detail in chapter 9 HIERARCHY-AWARE MEASUREMENT.

7.2.3.1. Foundations of Extensive Measurement

Arising from study of mass, a utility function to preserve combinations of objects needs to be identified. This means that we seek conditions on the relational system $(A, R, \circ)$ that are sufficient for the existence of a real-valued function $f$ on $A$ satisfying either the equations

$$aRb \iff f(a) > f(b) \text{ and } f(a \circ b) = f(a) + f(b)$$

or the equations

$$aRb \iff f(a) \geq f(b) \text{ and } f(a \circ b) = f(a) + f(b)$$

In other words, the goal is to identify conditions on $(A, R, \circ)$ being sufficient for the existence of a homomorphic map into $(\mathbb{R}, >, +)$, where $A$ is the underlying set, $R$ is a binary relation and $\circ$ is a binary operation [Rob79].

The definition of the Archimedean ordered group is closely related to the sufficient conditions for extensive measurement introduced by Hölder in 1901. In algebra a group is defined on $(A, \circ)$ fulfilling the axioms of associativity, of having an identity element and an inverse element. Based on the definition of the Archimedean ordered group presented in detail in [Rob79][pp.122-126], Hölder concluded (Hölder Theorem) that every Archimedean ordered group is homomorphic to $(\mathbb{R}, >, +)$. However, not each of the axioms of the Archimedean ordered group are necessary for identifying necessary and sufficient conditions for extensive measurement. Roberts
analyzed the necessity of the \textit{Archimedian ordered group} axioms for extensive measurement and identified four axioms covering the necessary and sufficient conditions for extensive measurement:

\textbf{E 1} (Weak Associativity). \textit{For all } \(a, b, c \in A\),
\[ [a \circ (b \circ c)]E[(a \circ b) \circ c] \]

\textbf{E 2}. \((A, R)\) is a strict weak order

\textbf{E 3} (Monotonicity). \textit{For all } \(a, b, c \in A\),
\[ aRb \iff (A \circ b)R(b \circ c) \iff (c \circ a)R(c \circ b) \]

\textbf{E 4} (Archimedean). \textit{For all } \(a, b \in A\), \textit{if } \(2aRa\) then there is a positive integer \(n\) such that \(naRb\)

Unfortunately, the given axioms together are not sufficient for extensive measurement. To obtain necessary and sufficient conditions for an extensive structure the last axiom is substituted by:

\textbf{E’ 4} (Archimedean). \textit{For all } \(a, b, c, d \in A\), \textit{if } \(aRb\) then there is a positive integer \(n\) such that \((na \circ c)R(nb \circ d)\)

A relational system \((A, R, \circ)\) satisfying the axioms E1, E2, E3 and E’4 is called an \textit{extensive structure}. Further, Roberts and Luce [RL68] identified that the relational system \((A, R, \circ)\) has an extensive structure if the following theorem holds:

\textbf{Theorem 7.2.1} (Roberts and Luce 1968). \textit{Suppose } \(A\) \textit{is a set, } \(R\) \textit{is a binary relation on } \(A\) \textit{and } \(\circ\) \textit{is a binary operation on } \(A\). \textit{Then there is a real-valued function } \(f\) \textit{on } \(A\) \textit{satisfying}
\[ aRb \iff f(a) > f(b) \text{ and } f(a \circ b) = f(a) + f(b) \]
\textit{if and only if } \((A, R, \circ)\) \textit{is an extensive structure.}

\textbf{7.2.3.2. Uniqueness}

The observations about measurement of physical properties such as mass suggests that the representation \(f\) should be unique up to a similarity transformation. In other words, measurement should be on a ratio scale.
7.3. Measurement Theory based Definitions of Software Measures/Metrics

**Theorem 7.2.2.** Suppose $A$ is a non-empty set, $R$ is a binary relation on $A$, $\circ$ is a (binary) operation on $A$, and $f$ is a real-valued function on $A$ satisfying 

$$a R b \iff f(a) > f(b)$$

and

$$f(a \circ b) = f(a) + f(b).$$

Then $U = (A, R, \circ) \rightarrow \mathfrak{B} = (\mathbb{R}, >, +)$ is a regular representation and $(U, \mathfrak{B}, f)$ is a ratio scale.

As result of this theorem identifying a relational systems $(A, R, \circ)$ as having an extensive structure implies that it is a *ratio scale*.

7.2.3.3. Additivity

Before closing our considerations on the application of measurement theory for software measurement we turn our attention to multiplicative representations. That is, if $f$ satisfies $a R b \iff f(a) > f(b)$ and $f(a \circ b) = f(a) + f(b)$ then $g = e^f$ satisfies $a R b \iff g(a) > g(b)$ and $g(a \circ b) = g(a)g(b)$.

Thus a multiplicative representation can also be obtained with positive $g$, then $f = \ln g$ gives an additive representation. The logarithm of a multiplicative representation gives rise to the same type of scale as the additive representation. In other words, the same comparisons using $\ln g$ are as meaningful as they would be using $f$.

7.3. Measurement Theory based Definitions of Software Measures/Metrics

In the previous sections we identified properties a homomorphism $f$, mapping an observed real world phenomena - represented by a relational system $U$ - to a set of (not necessarily) numerical values - represented by a numerical relational system $\mathfrak{B}$ -, should have. However, so far we did not go further into detail about a homomorphism $f$ being a measure or metric or if issues of relational systems, $f$ is depending on, need to be taken into account as well. Therefore, we present a definition of the terms measure/metric based on “Groubstake group’s” definition of these terms addressing the previously introduced terms and insights from measurement theory.
7. Software Measurement

7.3.1. Software Metrics

In the “Groubstake group’s” definition of a measure, the fourth property - “[…] identification and/or definition of mappings from the models to number systems which preserve the order relationship” - is used to define the term metric. By interpreting a homomorphism \( f \) as the mapping from an observed real world phenomena - the model - to a numerical system, several issues of measurement theory would directly and indirectly be affected by this interpretation.

A homomorphism \( f \) is defined as a mapping from one relational system to another. In the “Groubstake group’s” definition of a metric, however, neither the model - the origin - nor the number system, the model’s “attributes” are mapped to, are further defined. However, defining a homomorphism \( f \) to be a metric, further restrictions of the “model” and the “number system” as used in the “Groubstake group’s” definition of a metric need to be made. Based on the definition of a homomorphism, both, the “model” and the “number system”, are required to be relational systems. This, however, implies knowledge about relations between the model’s attributes as well as knowledge about operations that can be performed on the model’s elements.

Further, the “Groubstake group’s” definition of a metric requires order preserving mappings from a “model” to a “number system”. This implies knowledge about order relations between the “model’s” attributes as well as about order relations between the element’s of the number system. As introduced in section 7.2.1.1 Properties of Relations, different kind of order relations exist. The kind of the “model’s” underlying order relation is left open. However, the number system’s underlying order relation needs to correspond to the model’s one. It is not required that these need to be equivalent as, for instance, the quasi order’s underlying properties correspond to the partial order’s underlying properties except that the partial order has an additional property - the antisymmetry.

Operations, such as addition, of the elements of the model or of the elements of the number system may exist and are addressed by definition of relational systems. However, the “Groubstake group’s” definition of a metric does not address operations between the basic elements. Therefore, operations between the basic elements are not further considered in the definition of the term metric. As a consequence the relational systems representing both the “model” and the number system are
restricted to the set of elements and the relations between the basic elements. The relational system $\mathcal{U}$ is used to represent the model, $\mathcal{B}$ instead is used to represent the number system.

Having identified the relational systems $\mathcal{U}$ and $\mathcal{B}$ with the aforementioned properties, a homomorphism $f$ representing an order preserving mapping from $\mathcal{U}$ to $\mathcal{B}$ needs to be identified next. The “Groubstake group’s” definition of a metric only requires a mapping from the model to the number system to be order preserving. This means that it is sufficient to define any homomorphism $f$ and to analyze if the resulting triple $(\mathcal{U}, \mathcal{B}, f)$, the scale type, is order preserving. As introduced in section 7.2.2.4 SCALE TYPE several scale types exist. Ordering the presented scale types from weakest (nominal scale) to the strongest scale (absolute scale), the ordinal scale is the weakest scale with an order preserving mapping from $\mathcal{U}$ to $\mathcal{B}$. Therefore, we demand the scale $(\mathcal{U}, \mathcal{B}, f)$ to be at least an ordinal scale.

Based on the observations and interpretation of the “Groubstake group’s” definition of a metric towards the terminology of measurement theory we strengthen and slightly modify their definition of a metric and use the following instead:

**Definition 5 (Software Metric).** A metric is a homomorphism $f$ with an order preserving mapping of the elements of a relational system $\mathcal{U}$, with at least order relations being $\mathcal{U}$’s underlying relations, to the elements of a numerical relational system $\mathcal{B}$, with a corresponding underlying order relation. Further, the triple $(\mathcal{U}, \mathcal{B}, f)$ representing the metric’s scale type is at least an ordinal scale.

### 7.3.2. Software Measures

The “Grubstake group” defines a metric on the basis of a single one of a measure’s properties. Additional requirements for a measure address the relations and orderings between the elements of the model, which need to be determined by the models attributes. This implies that important relations between the elements need to be identified first. In contrast to the definition of a metric where the identification of order relations is required, in case of a measure other kinds of important relations between the elements need to be identified as well, e.g. relations based on operations, like addition, that can be performed on the elements of the model. Whether these are represented by relations, like the order relations in a relational system,
or if these are defined as basic attributes of the relational system’s elements is left open.

The first and second of the “Groubstake group’s” definition of a measure do not directly address properties covered by measurement theory and are more general. They mainly focus on the definition of criteria/attributes that need to be taken into account in the definition of a numerical representative of a software process/entity. This criteria can either be general concepts like quality characteristics such as usability, reliability and so on, or even more fundamental properties such as the Lines of code (LOC) or the number of methods.

For the definition of measures as representatives for quality characteristics, the basic definition of a quality characteristic needs to be analyzed and basic attributes need to be identified. Further, a combination of numerical representatives of the quality characteristic's basic properties towards a numerical representation of the quality characteristic itself needs to be defined. It needs to be ensured that existing (order) relations between the quality characteristics are represented by its numerical representatives as well.

Based on the interpretations of the “Groubstake group’s” definition of a measure we define a software measure as follows:

**Definition 6** (Software measure). A software measure is defined to be a numerical representative of attributes of a software product or process. A model capturing the attributes of a software product or process builds the basic set of a relational system $U$. Relations between the attributes are represented either as relations or as further attributes. Then a homomorphism $f$ is defined to be a measure if it is an order preserving mapping from the relational system $U$ to the elements of a numerical relational system $B$, with a corresponding underlying order relation. The triple $(U,B,f)$ representing the metric’s scale type is at least an ordinal scale.

### 7.4. Properties of Software Measures

In section 7.2 *Measurement Theory* we gave an overview of the basic principles of measurement theory as presented by Roberts in [Rob79]. The interpretation and adaptation of concepts and principles of measurement theory to software measurement by Bollmann-Sdorra and Zuse has lead to a controversially discussion
between Konrad [Kon91, Kon92] and Bollmann-Sdorra/Zuse [ZB89, ZBS92]. Generally, the adaptation of concepts and principles of measurement theory to software measurement presented by Zuse and more basically by the “Grubstake group” have been important contributions for formalizing software measurement. A different step towards a formalization of software measures is applied in [Wey88]. Weyuker identified nine basic properties for complexity of software entities. Beside describing the intuition of a particular property, Weyuker introduces a more formally representation of the properties, as well.

Based on the definition of software metrics and software measures introduced in the last section we turn our attention to the development of a method for the theoretical analysis of software measures and metrics.

The newly introduced definitions of both terms already cover several of the properties identified in Weyuker’s analysis of complexity. For the presentation of the properties based on Weyuker’s analysis we use the term software measures based on the definition of a metric presented in the ISO/IEC Std. 14598 [ISO99].

We first introduce properties being of interest for the theoretical validation of software measures for hierarchical dependencies based on Weyuker’s analysis of complexity of “software elements”. Additionally, properties not being addressed by Weykuer’s properties of complexity are identified and presented afterwards. At the end of this section we give an overview of properties used as foundation for a method to theoretically validate software measures for hierarchical dependencies.

### 7.4.1. Fundamental Properties of Complexity

In [Wey88] Weyuker introduced basic properties for complexity in both a formal and an informal way. For each of the properties a textual description introduces its intuition followed by a formal, mathematics based interpretation of the intuition. Additionally, she analyzed well known software measures, like McCabe’s cyclomatic complexity [Mcc76] and Halstead’s effort [Hal77] to determine whether these are able to fulfill the introduced properties.
7. Software Measurement

7.4.1.1. Unconformity between the Intuition and its Formalization

Some of the formal definitions presented by Weyuker do not exactly reflect the intuition of the property. However, the intuitions itself are highly valuable, as they give a clear impression of what the concept of complexity of software entities is about.

In the following we show that the formalized description of the second and third of Weyuker’s properties of complexity do not exactly correspond to their informal intuition by use of a simple real world example.

The second property of complexity of software elements addresses the intuition “[...] that a measure is not sensitive enough if it divides all programs into just a few complexity classes”. This intuition has been formalized as:

→ “Let c be a nonnegative number. Then there are only finitely many programs of complexity c”.

The third property addresses the opposite intuition, that it is unlikely that “[...] rating too many programs as being of equal complexity, we do not want a measure to be too “fine” and assign to every program a unique complexity”. This is formalized as:

→ “There are distinct programs P and Q such that |P| = |Q|”.

The formalized description of the first three of Weyuker’s properties do not correspond to the intuition as can easily be seen by the following example. Instead of using the complexity of software elements we apply these properties to the mass of objects, as these properties are not restricted to complexity and are valid for measures of other fields as well.

As the objects to be measured the mass of pieces of paper in the DIN A4 format is used. Depending on the piece of paper’s thickness described as g/m² different masses can be measured. A DIN A4 piece of paper of 80 g/m² has a mass of approx. 5.0 g, a DIN A4 piece of paper with 100 g/m² has a mass of 6.2 g whereas a DIN A4 piece of paper with 250 g/m² has a mass of approx. 15.6 g.

A set \( B \) being a subset of \( \mathbb{R}^+ \) where the values do not have more than two decimal places is used to give a numerical representation of the piece of paper’s mass - for example a balance with an accuracy of 0.01 kg is used to weight (measure) the piece of papers mass. \( A \) represents the mass in g. An operation \( f \) represents the mapping between the observed mass represented by the relational system \( \mathcal{U} \) to the numerical relational system \( \mathcal{B} \) with \( B \) as its underlying set. All of the formalized properties are fulfilled by the scale \( (\mathcal{U}, \mathcal{B}, f) \) : There are only finitely many types of pieces of paper masses that are mapped to a particular value of \( B \) - e.g. 0.01 or 0.02, so that the first property is fulfilled. The second and third formalized property are fulfilled as well, as there are distinct masses of pieces of paper such that \( f(a_1) = f(a_2) \) (property 3) and there exists a \( f(a_1) \neq f(a_2) \) (property 2).

However, the basic intuition that too many pieces of paper are of equal mass does not hold when \( B \) is used as the underlying set of the numerical relational system \( \mathcal{B} \), as basically the
mass is only mapped to one of two values of \( B, 0.01 \) and \( 0.02 \). From a measurement theory's point of view, \( B \) is not appropriate either, as the observed relations between the different piece of papers mass are not reflected by \((\mathbb{U},\mathbb{B},f)\).

In the given example we addressed the unconformity of the intuition of the second and third of Weyuker's complexity properties and its formalization. We made use of basic concepts of measurement theory and a simple real world example to show the unconformity between a property's intuition and its formalization. Although an unconformity exists, the basic intuitions are still of importance. In addition both analyzed properties, the first and eighth property of complexity measures address a restriction of the range of adoptable values by a complexity measure, as well. In terms of measurement theory, these properties address the identification of an appropriate scale \((\mathbb{U},\mathbb{B},f)\) mapping a real world phenomena to a numerical relational system. This, however, is not particular to complexity measures. A generalization of the property's intuition is valuable for measures addressing other areas as well and is analyzed in the following.

### 7.4.1.2. Generalized Properties

Having identified that the formalization of Weyuker's properties of complexity measures does not conform to the intuition in the previous section, we turn our attention to the generalizability of the properties. We focus on the properties' intuition and identify concepts and definitions of measurement theory that are addressed by the intuitive definition of a property.

The first three properties and the eighth property proposed in [Wey88] are related to the range of values a complexity measure should adopt. The first property addresses the fact that it is unlikely that a measure rates all software entities as being equally complex. She strengthens this property by the second one stating that it is also undesirable that a complexity measure is not sensitive enough, e.g. if it divides all software entities into few complexity classes [Wey88]. Finally, the third property and its strengthening the eighth property have its focus on the intuition that it is undesirable that the complexity of software entities are mapped onto a “too fine grained” set. The eighth property strengthens this intuition as it addresses the fact that not each software entity should be mapped to an individual number, which
Software Measurement

would lead to an identification of the software entities through these numbers and would not represent the software entities’ complexity.

These properties directly and indirectly address several concepts of measurement theory. Although not directly related to measurement theory, the identification of relations addressing a particular purpose e.g. complexity between objects is a basic concept of measurement theory. However, beside the identification of relations, the objects these are performed on need to be identified as well. In the case of complexity measures, the set \( A = (a_1, a_2, \ldots, a_n) \) represents software entities. Typical examples of relations between the objects of \( A \) are “\( a_p \)’s complexity is higher than \( a_q \)’s complexity (H)”, “\( a_p \)’s complexity is higher or at least as high as \( a_q \)’s complexity (HE)”, and so on. More generally, this leads to a relational system \( \mathcal{U} = (A, R) \), with \( R \) defining a relation between the objects of \( A \). Assuming a relational system \( \mathcal{U} = (A, HE) \), then an appropriate numerical relational system, \( \mathcal{U} \) could be mapped to, is \( \mathcal{B} = (B, \geq) \).

The intuition behind the second, third and eighth property of complexity measures has an impact on the constitution of \( B \) and the homomorphism \( f \) - proposing that \( B \) should consist of more than a single value is not sufficient, as it is still possible that \( f \) would map all objects of \( A \) to a particular element of \( B \). The identification of a numerical relational system \( \mathcal{B} \), being appropriate to represent concepts such as complexity of real world concepts represented by \( \mathcal{U} \) is not straightforward, as shown in the previous example. Before we can turn our attention to the homomorphism \( f \) an appropriate \( \mathcal{B} \), corresponding to the real world observation and relations between objects represented by \( \mathcal{U} \), needs to be identified. For instance, to measure the mass of pieces of paper (see previous example) it would be sufficient to choose a subset of \( \mathbb{R}^+ \) as the underlying set of \( \mathcal{B} \) if the homomorphism \( f \) is fine grained enough, like a mapping of the masses to a relational system being a representation of mass in gram. Further, it needs to be identified how the observed relations between real world objects can be represented by \( \mathcal{B} \). For instance, relations between the masses of pieces of paper are “\( a_p \)’s mass is higher than \( a_q \)’s mass (H)”, “\( a_p \)’s mass is higher or at least as high as \( a_q \)’s mass (HE)”, and so on. In \( \mathcal{B} \) these are represented by
More generally, it is necessary to identify an appropriate numerical relational system \( \mathfrak{B} \) corresponding to \( \mathfrak{U} \).

\[ \mathfrak{U} \hookrightarrow \mathfrak{B} \implies \text{Identification of an appropriate numerical relational system } \mathfrak{B} \text{ corresponding to } \mathfrak{U}. \]

Beside the identification of an appropriate numerical relational system \( \mathfrak{B} \), a homomorphism \( f \) representing the mapping of elements of \( A \) into \( B \) needs to be identified. Similar to the identification of \( \mathfrak{B} \), \( f \) needs to be appropriate. Inappropriate mappings are directly addressed by the second, third and eighth of Weyuker's properties. An inappropriate \( f \), for example, maps each element of \( A \) to a single element of \( B \). However, \( f \) would also be inappropriate if it would map each element of \( A \) to a unique element of \( B \), where the value an element is mapped to could be used as the identification of the object, like the number on the jerseys of football players. Therefore, the corresponding scale type - the nominal scale - is inappropriate for software measures. The fifth of Weyuker's properties addresses the intuition that “monotonicity” is another fundamentally important property for complexity measures. “Monotonicity” is not particular to complexity measures and software measures should generally be monotonic transformations.

In other words, the scale \( \mathfrak{B} \) at least needs to be unique up to order implying the admissible transformations from \( \mathfrak{U} \) into \( \mathfrak{B} \) being the class of monotone increasing functions \( f(x) \). As presented in the previous section the corresponding scale type is the ordinal scale.

\[ \mathfrak{U} \hookrightarrow \mathfrak{B} \implies \text{Identification of an homomorphism } f \text{ - representing a measure - mapping } \mathfrak{U} \text{ to } \mathfrak{B} \implies (\mathfrak{U}, \mathfrak{B}, f). \]

However, the possibilities for applying statistical methods to ordinal scales are restricted. Beside variance, standard deviation, mean and other basic statistical methods, usually dependencies between the software measures/metrics need to be analyzed. These dependencies can be obtained by performing a correlation analysis, for instance, with either the Spearman rank correlation coefficient or the Bravais-Pearson correlation coefficient. A pre-requisite for Spearman’s rank correlation coefficient, however, is an ordinal scale, whereas for the Bravais-Pearson correlation coefficient at least an interval scale is required. For the calculation of the variance and standard deviation an interval scale is assumed as well. Basically the scale
(U, S, f) at least being an ordinal scale would be sufficient. However, for the identification of dependencies between software measures/metrics and the application of basic statistical methods at least an interval scale is required.

Identification of the underlying scale type of (U, S, f). An ordinal scale is sufficient, whereas an interval scale would be preferable, for the application of statistical methods.

The fourth of Weyuker’s properties is a strengthening of the first property of complexity measures. In contrast to the first three properties addressing the range of adopted values, the fourth property takes the syntax and semantics of program bodies a measure is performed on into account. It assumes that two implementations of the same functionality may have a different complexity. A measure depending on the implementation rather than the functionality it addresses can be thought of as an abstraction of this property.

Identification if a measure addresses the implementation of a functionality or it addresses the functionality itself.

7.4.1.3. Particular Complexity Properties

In property five and its strengthening property nine, Weyuker generally assumes that “program bodies” can be concatenated. However, she does not go into further detail how a composition of software entities is performed. In the fifth property she only assumes the complexity of a concatenated software entity \( f(P, Q) \) - with \( f \) representing a complexity measure - at least needs to be as high as the highest complexity of its basic elements \( f(P) \) or \( f(Q) \). Taking side-effects between the concatenated “program bodies” into account she proposes that the complexity of the composed “program bodies” \( f(P, Q) \) should be higher than the sum of its basic elements.

In [ZB89], Bollmann-Sdorra and Zuse use a different assumption for the complexity of concatenated software entities. They require complexity measures to be additive, that is, the complexity of concatenated software entities can be obtained by summing up the complexity of its basic software entities: \( f(P, Q) = f(P) + f(Q) \). However, in contrast to Weyuker, directly relating the analysis of complexity to software
entities itself, Bollmann-Sdorra and Zuse use an abstraction of the software entities, instead, by making use of a graphical representation of the software entities, namely a control flow representation, for their analysis. Applying the analysis of concatenated operations to concatenated control flow graphs rather than concatenated operations of software entities, possible interactions between operations as proposed by Weyuker's properties five and nine are not addressed. Further, requiring complexity measures to be additive strengthens the negligence of interactions between concatenated operations.

Taking the interactions between concatenated operations into account, software measures addressing the concept of complexity can not be additive. Whether interactions between e.g. concatenated operations have an influence on other concepts than complexity such as reliability, maintainability and so on, individually needs to be analyzed for a particular concept. Anyway, being able to make use of measures being additive for a particular concept, has the big advantage that in the case of measuring hierarchical dependencies we would be able to apply the measure on the individual operations and obtain the aggregated value by summing up the individual values.

→ **Identification if a particular measure fulfills the property of being additive.**

Whether concepts like the quality characteristics are additive or an influence between basic concepts these are measured on exists, needs to be discussed individually for a particular concept as well. This, however, is not in the focus of our recent work and is part of our further research on this topic.

Similar to the fifth and the ninth of Weyuker's properties addressing the interactions between concatenated operations, the sixth and seventh property more basically address the interactions between statements within the program bodies itself. Whether interactions between statements within program bodies are of importance for a particular measure or concept individually needs to be decided.

→ **Identification if interactions between statements and/or concatenated operations affect a particular measure.**
7. Software Measurement

7.4.1.4. Viewpoints and Sub-Concepts

In [ZB89] Bollman-Sdorra and Zuse realized that each “software complexity measure is connected with an (mostly unknown) idea of complexity” [ZB89]. The idea of complexity is then called a “viewpoint”. Additionally, the concept of an “elementary viewpoint” is introduced as being a reduction and description of a “viewpoint” (see [ZB89, p.26] for details).

The introduction of “viewpoints” for complexity measures, as proposed by Bollmann-Sdorra and Zuse, has lead to heavy criticism by Konrad [Kon91]. Konrad based his criticism amongst others on the use of the term “viewpoint”. He recognized that the term “viewpoint” is used with two different meanings describing both “a relation between objects and as a set of beliefs about objects” [Kon91]. More generally, he criticized the misuse of a “change a sound and productive method for information retrieval” - the use of user viewpoints as proposed by Cherniavsky and Lakhuty [CL70] - “into a speculative trick for software metrics” [Kon91]. Further, Bollmann-Sdorra and Zuse, try to combine the concepts of a “viewpoint” and a metric, as they introduce the concepts “elementary viewpoint of a metric” and “viewpoint of a metric” [Kon91]. However, Konrad has shown that neither an “elementary viewpoint” nor an “elementary viewpoint of a metric” nor a “viewpoint of a metric” are a “viewpoint” at all and that “viewpoints” allow to corroborate empirical statements by pure speculation [Kon91, p.55-p.56].

Bollmann-Sdorra and Zuse combined the basic concepts of measurement theory as introduced by Roberts [Rob79] and Krantz et al. [KLST71] with the newly introduced concept of “elementary viewpoints” to obtain a set of criteria for scale types. Based on these criteria, the commonly used complexity measure of McCabe [Mcc76] - the cyclomatic complexity - has been determined to fulfill the criteria of an ordinal, interval or ratio scale. Additionally, Bollmann-Sdorra and Zuse analyzed the concatenation of operations (on control flow graphs) and applied a modified set of axioms to show that the modified McCabe metric \((\sigma - 1)\) implies an extensive structure [Kon91, p.61].

In contrast to viewpoints, as introduced by Bollmann-Sdorra and Zuse, sub-concepts - in the ISO/IEC 9126 the term sub-characteristic is used - address the fact that a general concept (quality characteristic) such as reliability is comprised of sev-
eral sub-concepts that need to be taken into account when a measure is developed for these. The partitioning of quality characteristics in sub-characteristics has already been introduced in the late 1970’s by Boehm [BBL76] and McCall [MRW77]. The partitioning of characteristics in sub-characteristics has been commonly accepted over the last recent years and the standard ISO/IEC 9126 [ISO91] is based on the results of these publications. It makes use of a partitioning of the seven main-characteristics in a similar but advanced manner.

7.4.2. Properties based on Graphical Abstractions

As discussed in the previous section it is possible that by using a graphical abstraction as the basis for the analysis of software measures important dependencies between implementations (e.g. operations or statements) are not taken into account. In other words, before using a graphical representation, such as control flow graphs rather than the implementation itself, it needs to be analyzed whether particular properties are addressed by the graphical representation as well. In case of complexity, using a control flow graph as representation of the implementation the analysis is performed on - the representation of a software entity Bollmann-Sdorra and Zuse used for their analysis - dependencies between statements may unintentionally not be considered.

Currently, in the field of software engineering the UML 2.x [Obj09b] is often used to model complex applications based on a graphical representation. The UML provides thirteen diagram types in total, each of which is a specialized graphical representation of the different facets of an application. Generally, the diagram types are grouped in those addressing the structural properties of an application such as the class, object, component and composition diagrams, those addressing the dynamic properties like activity and state-chart diagrams and those being further grouped as interaction diagrams like sequence, communication and timing diagrams.

In the first part we identified KobrA as of being the only component based methodology with an underlying view centric platform independent component model. Basically, KobrA makes use of different views to describe the facets of a component such as the structural, behavioral and functional view of a component’s specification and realization. Generally the views are modeled by use of different diagram
7. Software Measurement

types of the UML as presented in section 4.4 KOBRA. Although Kobra has primarily been designed for the development of component based systems, highlighting different facets of a component by the use of particular views, it is not restricted to components at all. More generally, it is possible to make use of views represented by particular UML diagram types to model the facets of software entities as well.

Generally, it is possible to model software entities at different levels of detail using UML. Activity and sequence diagrams allow modeling software entities up to a very low level of detail, like modeling branches or loops. However, activity diagrams, e.g. in Kobra, are used to model algorithms, whereas sequence diagrams traditionally are used to model interactions between an actor and a software entity - the system - on a very high level of detail, where the term system sequence diagrams is used [Lar04]. Whether, issues like interference between statements, which need to be taken into account for complexity measures, are visible in the graphical representation using activity, sequence or other UML diagram types depends much more on the level of details of the model.

7.4.3. Properties based on the Software Metric’s/Measure’s Definition

The properties identified in the previous section are based on the definition of software measures/metrics presented in the ISO/IEC Std. 14598 [ISO99]. Several properties that have been identified as being of importance in the previous section are also covered by the newly introduced definition of software metrics/measures. In the following we focus on the identification of properties related to our definition of software measures/metrics. Then, in the next section both sets of properties are combined and a method for the theoretical analysis of software measures/metrics is introduced.

A basic issue of the measure’s definition is the identification of a relational system $U$ representing the model’s attributes and order relations defined on these.

$\leadsto$ **Identification of attributes of the underlying model being of importance. These build $U$’s underlying set $A$.**

The definition of the term metric does not require the identification of operations that are applicable to the model’s basic elements. This means that for the identi-
7.4. Properties of Software Measures

Identification of order relations $R_1, \ldots, R_n$ between the elements of $A$.

The combination of set $A$ and the order relations $R_1, \ldots, R_n$ identified as of being of importance for the elements of $A$, results in the definition of the relational system $\mathcal{U} = (A, R_1, \ldots, R_n)$.

Definition of the relational system $\mathcal{U}$ based on the set $A$ and the relations $R_1, \ldots, R_n$ between $A$'s elements.

Software measures/metrics are basically thought as of being order preserving mappings from a model to a number system. Therefore, it is necessary to identify a numerical relational system $\mathfrak{B}$, being appropriate and corresponding to the relational system $\mathcal{U}$.

Identification of an appropriate numerical relational system $\mathfrak{B}$.

The mapping from $\mathcal{U}$ to $\mathfrak{B}$ is defined to be order preserving. As a consequence, it is necessary to identify if a particular homomorphism $f$, mapping the elements of $A$ to $B$ is order preserving. This implies that the scale defined by $(\mathcal{U}, \mathfrak{B}, f)$ at least needs to be an ordinal scale.

Identification if the triple $(\mathcal{U}, \mathfrak{B}, f)$ is at least an ordinal scale.

The properties identified so far, are related to the definition of a metric. The definition of a measure further requires the identification of important relations between the elements of the underlying model. This means, that relations addressing the application of operations on the elements of $A$ need to be considered as well. This implies that both applicable operations on the model's basic elements as well as relations and properties resulting from the application of the operations need to be identified.

Identification of applicable operations on the models elements.

Identification of additional relations and properties resulting from the identification of applicable operations on the model's elements.
7. Software Measurement

7.5. Method for the Theoretical Analysis of Software Measures

In section 7.4.1 FUNDAMENTAL PROPERTIES OF COMPLEXITY we discussed the relation between Weyuker’s properties of complexity and measurement theory as introduced by Roberts. Further, we identified properties resulting from the newly introduced definition of software measures/metrics in the previous section. In the following we turn our attention on defining a method for the theoretical analysis of software measures/metrics for hierarchical dependencies. The introduced method, however, is not restricted to these measures and may be of value for a different context as well.

Basically, the identification of attributes of the underlying model - the basic set \( A \) of the relational system \( \mathcal{U} \) - of importance is the initial step in the theoretical analysis of software measures. Based on the elements of \( A \), order relations between the elements of \( A \) need to be identified afterwards. This, however, implies the identification of basic properties of the analyzed software measure/metric such as the kind of symmetry, reflexivity and so on, in order to identify the underlying kind of order.

**Step 1.** Identification of the basic set \( A \) of the relational system \( \mathcal{U} \).

**Step 2.** Identification of order relations \( R_{\mathcal{U}} = R_1, \ldots, R_n \) between the elements of \( A \).

**Step 3.** Identification of basic properties like symmetry, reflexivity, transitivity and completeness of the relations of \( R_{\mathcal{U}} \).

**Step 4.** Identification of the relations’ \( R_1, \ldots, R_n \) kind of order.

Based on the results of the analysis of Weyuker’s properties and the identification of properties resulting from the definition of software measures/metrics in the previous section, in the next step it is necessary to identify if the numerical relational system \( \mathcal{B} \) is appropriate. Therefore, resulting from the analysis of Weyuker’s properties of complexity, it is necessary to analyze the range of adopted values of \( B \). As we have shown neither should a mapping from the elements of \( A \) to \( B \) result in a too insensitive mapping of the elements of \( A \) to a small number of values, nor should the homomorphism \( f \) be a too sensitive mapping of the elements of \( A \) to individual numbers. Additionally, it is necessary to identify if \( \mathcal{B} \)’s relations type order corresponds to the one of the respective relation in \( \mathcal{U} \).
7.5. Method for the Theoretical Analysis of Software Measures

**Step 5.** Identification if the set \( B \) is appropriate for the elements of \( A \) to be mapped to.

**Step 6.** Identification of relations \( R_B = R'_1, \ldots, R'_n \) corresponding to the relations of the relational system \( \mathcal{U} \).

**Step 7.** Identification of basic properties like symmetry, reflexivity, transitivity and completeness of the relations in \( R_U \).

**Step 8.** Identification of the relations' \( R'_1, \ldots, R'_n \) kind of order.

**Step 9.** Determination of whether the kind of order of \( R'_m \) corresponds to the one of \( R_m \).

Basically, operations, like concatenation operations, are thought as of being applicable on the elements of \( A \). Before it is possible to identify corresponding operations applicable on \( B \), it is necessary to identify the operation’s properties. As discussed for complexity, Weyuker for instance, assumes that interactions between the statements of software entities result in a complexity of the concatenated element being higher than the one of its basic elements. In contrast, Bollmann-Sdorra and Zuse neglect interactions and assume that the complexity of the concatenated element is the sum of its basic elements. In other words, they assume the concept of complexity being additive. As well as, identifying operations that are applicable to the basic elements, it is necessary to identify if particular interactions, side-effects and so on exist and need to be taken into account for a particular operation. Although other operations may be performed on the basic elements, concatenation is of most importance for hierarchical dependencies. Further, it is necessary to identify relations \( R_B \) corresponding to already identified relations \( R_U \). Taking these issues into account the next step of the theoretical analysis of composition hierarchies of component based systems are:

**Step 10.** Identification of operations applicable on the elements of \( A \) being of importance, like the concatenation operation \( \rightarrow \mathcal{U} = (A, R_U, \circ_U) \).

**Step 11.** Identification of the operations properties. For instance, if interactions between statements may need to be taken into account.

**Step 12.** Identification of corresponding operations applicable on the elements of \( B \), resulting in a numerical relational system \( \mathfrak{R} = (B, R_B, \circ_B) \).
7. Software Measurement

Basically, the identification of the scale’s $(U, \mathcal{A}, f)$ type is a pre-requisite for the application of statistical methods on empirical data. As discussed earlier, statistical methods such as variance and standard deviation are only applicable if a particular scale type is identified. For instance, variance and standard deviation are only applicable if an interval or ratio scale is identified. However, based on the newly introduced definitions of measures and metrics, it is sufficient if $(U, \mathcal{A}, f)$ is an ordinal scale.

**Step 13.** Identification of the scale’s $(U, \mathcal{A}, f)$ type, that at least needs to be an ordinal scale.

At least, it needs to be identified if a homomorphism $f$, the relational system $U$ representing the basic attributes of the model to be measured and the numerical relational system $\mathcal{A}$ these are mapped to, fulfill the properties of a measure/metric.

**Step 14.** Identification if $f$, $U$ and $\mathcal{A}$ fulfill the properties of a measure/metric.

7.6. Summary

In this chapter we gave an overview of the basic principles of measurement theory that are used for a theoretical analysis of software measures addressing the hierarchical dependencies in composition hierarchies. The introduced properties, however, are not particular, to these systems and are generally applicable. The basic principles introduced are mainly based on the concepts and principles presented by Roberts in [Rob79].

In the subsequent section we analyzed Weyuker’s properties for complexity measures and successfully gave an abstraction of the properties based on the previously introduced concepts of measurement theory. We also discussed the applicability of graphical representations such as control flow graphs that in the case of complexity - Bollmann-Sdorra and Zuse analyze complexity based on control flow graphs - have led to the issue that an interference between statements of a particular software entity was not recognizable. As UML diagram types such as activity or sequence diagrams allow software entities to be modeled at a very low level of detail, the representational gap between an implementation using a high level programming language like Java and a detailed model using particular UML diagram types is neglectable in the given context.
7.6. Summary

Further, we applied principles and terms of measurement theory to the definition of software measures/metrics introduced by the “Groubstake group” resulting in a measurement theory based definition of both terms. Basic properties were identified to build the foundation of a method for the theoretical analysis of software measures introduced in chapter 9 HIERARCHY-AWARE MEASUREMENT.
Software Metrics-Related Work

Dozens of publications introduce or discuss software metrics in recent years. Basically, the software metrics presented address a definite concept and reach from direct measures/metrics covering a particular concept like the well known LOC to software metrics addressing more complex concepts. Concepts that gained major interest and are widely discussed in the literature are the so called quality characteristics defined in the ISO/IEC 9126 [ISO91].

Six main characteristics (functionality, reliability, usability, efficiency, maintainability and portability) are defined for the internal/external quality. These are composed of two to four non-overlapping sub-characteristics. Beside a generic model the ISO/IEC 9126 describes how each of these main-/sub-characteristics can be measured by representative metrics [JKC04]. However, the ISO/IEC 9216 leaves it open in which context the metrics that are defined to be indicators for particular quality characteristics can be applied. Further, the ISO/IEC 9126 [ISO01] does neither discuss why a metric can be interpreted as of being an indicator of a particular quality characteristic, nor is it further discussed up to which degree a metric is able to give a numerical interpretation of a particular concept.

In practice, admittedly, it is often difficult to identify whether metrics can be interpreted as of being indicators of a particular concept. The aforementioned LOC, for
instance, is often interpreted as of being an indicator for the complexity of a software entity. Beside the quality characteristics, e.g. the ones defined in the ISO/IEC 9126, complexity is the concept most being discussed in the literature. Based on the underlying context several indicators for complexity are defined. Due to its importance we give a brief overview of indicator metrics for complexity in the following section.

Although we are aware of the importance of interpreting software metrics towards a particular concept like the quality characteristics, we focus on giving an overview of the most important metrics itself in this chapter rather than going more into detail on the interpretation of metrics towards a particular concept (except complexity). Beside single measures like the cyclomatic complexity often a set of metrics - a so called metric suite - addressing a particular concept or being defined for a particular context are introduced in the literature. The most important ones are the Halstead, the Object Oriented and the Metrics for Object Oriented Design (MOOD) metric suites.

Beside the Halstead metric suite being defined on textual documents, both the Object Oriented and the MOOD metric suite address the issues related to object oriented programming. The SDMetric suite, however, gives an overview of metrics addressing the basic concepts that are captured by UML diagrams in an isolated way. An overview of the SDMetric suite is presented afterwards in section 8.3 DESIGN MEASURES/METRICS. Finally, we go into detail on some metrics addressing software components. These mainly focus on the issues of the encapsulated entity rather than the composition hierarchy these are integrated in.

8.1. Traditional Complexity Metrics

In the literature a large number of metrics are discussed that address the concept of complexity based on the underlying representation of a software entity. The LOC, for instance, is one of the most discussed metrics that are thought of indicating the complexity of a source code representation of a software entity. Other metrics such as the cyclomatic complexity measure the complexity of a software entity based on a graphical representation of the statements of a source code representation and
8.1. Traditional Complexity Metrics

the transition from one statement to one another. Both, however, are inadequate to measure the complexity of software components.

Specialized metrics addressing the complexity of an isolated software component have been introduced in recent years, such as the Component Packing Density (CPD) or the Component Interaction Density (CID) [NH04]. These are discussed more in detail in section 8.4 Component Measures/Metrics. Complexity metrics based on the design of a software entity exist as well, such as the “number of messages sent to instances of the same class” defined in [LLWW95].

8.1.1. Lines of Code

The LOC metric is one of the simplest metrics addressing the complexity of a software entity from a source code perspective. The LOC is further used in conjunction with other metrics or time to indicate other concepts. Although, the LOC in conjunction with a time period, such as the LOC per person month indicating the productivity of a programmer, is less conclusive, it is still used in practice due to its easy applicability and understandability from a management perspective [Tha00].

Basically, different interpretations on how to count a particular line of code exist. In [Tha00] Thaller lists the following possibilities of counting a particular line of code in a source code representation of a software entity. The LOC is defined as either:

- counting only the lines with executable statements;
- counting the lines of code with executable statements and the definition of data types;
- counting the lines of code with executable statements, definitions of data types and comments;
- counting the lines of code with executable statements, definitions of data types and comments and additionally the related commands in a Job Control Language (JCL);
- counting the physical lines as these are found when editing with an editor;
- or at least by using the delimiters as basis for the LOC.
8. Software Metrics-Related Work

Depending on the selected definition the results gained by applying this metric vary. However, mostly more than a single statements is placed in a particular line in the source code representation. Thus, using the LOC as basis for the indication of productivity a programmer can manipulate the source code in a way that his productivity is maximized, for instance by writing a single statement in each line of his realizations.

8.1.2. Cyclomatic Complexity

In [McC76] McCabe rejected the LOC as of not being sufficient to indicate the complexity of a software entity as a relationship between the length and the module complexity was not seen. Basically, McCabe, intended to introduce a metric to give an estimation on the degree of a software entity's modularization, where the goal of McCabe was to identify the modules which would be difficult to test or maintain.

In contrast to the LOC that is directly applied on the source code representation of a software entity, the cyclomatic complexity aims to give a statement on the complexity of the software entity based on the control flow graph of the module. He decided to base his considerations on control flow graphs as he assumed it would reflect best the software entity's testing effort [McC76]. In practice often the source code representation is used as starting point, where the control flow graph is directly extracted from. Basically, the control flow graph can be separated in nodes corresponding to a block of code with a sequential flow and arcs that correspond to branches taken by the application.

The complexity is then defined as

\[ \vartheta(G) = E - N + 2, \]

where \( E \) is defined as the number of flow graph edges and \( N \) is given by the number of graph nodes.

An alternative for the calculation of the cyclomatic complexity is given by the equation

\[ \vartheta(G) = \pi + 1, \]
8.2. Metric Suites

where $\pi$ represents the number of predicate nodes contained in the flow graph $G$ [McC76].

The McCabe metric was subject to heavy criticism in the literature. In [She88] Shepperd realized that McCabe was not successful to provide a new metric for complexity based on the assumption of the LOC not being sufficient as indicator for complexity. In [WHH79] the authors analyzed 330 Fortran subroutines, where both the number of lines of code and the nodes (nodes and essential nodes) were calculated. The authors opposed the results of the empirical analysis of the LOC with the cyclomatic complexity of the given subroutine, where they found a strong correlation (Brevais-Pearson) between the LOC and $\vartheta(G)$ with a value of 0.98.

However, it should be noted that the cyclomatic complexity compared to the LOC is less manipulable. In the previous section we have discussed that for instance the productivity based on the LOC (LOC per person month) can be increased by placing a single statement in each line. This kind of manipulation of the source code representation has almost no influence on a productivity metric being defined on basis of $\vartheta(G)$ (e.g. $\vartheta(G)$ per person month).

8.2. Metric Suites

The traditional metrics presented in the previous section address the complexity of a software entity from the source code perspective and are defined as individual metrics. However, sometimes metrics are grouped in metric suites. One of the most important metric suites addressing several facets regarding the “source code” (textual) representation of a software entity is the Halstead metric suite.

With the appearance of object oriented programming several researchers analyzed whether the traditional metrics where able to address the structural aspects of object oriented systems [TSM92]. In [MD89] Moreau and Dominick identified that the traditional metrics were at least effective to address the content of an object’s operation. Anyway, the metrics being of major importance that address the paradigm of object orientation where mainly grouped in metric suites, like the Object-Oriented metric suite by Chidamber and Kamerer [CK94, CK91] or the MOOD metric suite by Brito e Abreu [BC94].
8. Software Metrics-Related Work

8.2.1. The Halstead Metric Suite

The Metric suite of Maurice Halstead [Hal77] is one of the most widely known metric suites in the literature and was introduced in the late 1970's. It consist in total of nine software metrics that can be categorized in basic metrics ($\eta_1, \eta_2, N_1, N_2$) and metrics addressing a more complex facet of a software entity.

8.2.1.1. Basic properties

The more complex metrics in the Halstead metric suite are based on the four direct metrics $\eta_1, \eta_2, N_1, N_2$ that have their focus on the basic structure of a software entity. These basically address a software entity’s underlying structure by use of its operators and operands:

$\eta_1$ represents the number of unique or distinct operators, where

$\eta_2$ represents the number of unique or distinct operands.

For both, $\eta_1$ and $\eta_2$, the unique and distinct number of operators and operands are added up, where $N_1$ and $N_2$ have their focus on the total number of operators and operands:

$N_1$ represents the total usage of all operators, where

$N_2$ represents the total usage of all operands

Two metrics were defined by Halstead that are directly based on the basic metrics. The Program Length ($N$) is defined as the sum of the total number of operators $N_1$ and operands $N_2$. The Program Vocabulary ($\eta$), instead, is defined as the sum of the unique or distinct operators $\eta_1$ and operand $\eta_2$.

**Program Length** $N = N_1 + N_2$

**Program Vocabulary** $\eta = \eta_1 + \eta_2$

8.2.1.2. Program Volume

In order to define a metric for the size of a software entity, Halstead introduced the Program Volume ($V$). The introduced Program Volume allows to compare software
fragments that cover the same kind of functionality, but that are defined in different programming languages.

\[ V = N \cdot \log_2 \eta \]

### 8.2.1.3. Difficulty

The next metric in the Halstead metric suite is concerned with the difficulty to write a particular piece of software or how difficult it is to understand the details of a software fragment, for instance when it is reused. Halstead, defines the **Difficulty** \((D)\) on basis of the basic metrics \(\eta_1, N_2\) and \(\eta_2\):

\[ D = \frac{\eta_1 \cdot N_2}{2 \eta_2} \]

### 8.2.1.4. Effort

The Halstead **Effort** \((E)\) metric describes the programming effort that is required to reduce a preconceived algorithm to an actual programming language [Hal77]. Further, in the literature the Halstead **Effort** is often denoted as of being an indicator for the complexity of a software entity in the literature, for example in [Wey88] and [TSM92].

The relationship between the metrics described earlier in this section and the effort is explained by Halstead in six simple and straight forward steps. We only present the resulting equation, for more details the interested reader is referred to [Hal77].

\[ E = D \cdot V \]

### 8.2.2. The Object Oriented Metric Suite

The Halstead metric suite presented in the previous section has it focus on determining particular facets of a source code (textual) representation of a software entity based on the procedural programming paradigm. The **Object oriented (OO) metric suite** introduced by Chidamber and Kemerer [CK91, CK94], instead, has its focus on providing metrics to cover the basic principles of object oriented development defined as a general idea by Bunge in [Bun77, Bun79].
8. Software Metrics-Related Work

8.2.2.1. Weighted Method per Class

The Weighted Method per Class (WMC) indicates the weighted complexity of a single class based on the complexity $c_i$ of its underlying methods:

$$\text{WMC} = \sum_{i=1}^{n} c_i$$

In [CK94], Chidamber and Kemerer, do not refer to any particular complexity metric. Thus, the choice which complexity metric $c_i$ is applied to measure the WMC is left to the user.

8.2.2.2. Depth of Inheritance Tree

The Depth of Inheritance Tree (DIT) is the second metric defined in the OO suite. The DIT measures what Bunge defined as the scope of properties. This metric is defined as the maximum length from a particular node to the root of the inheritance tree (see Figure 8.1), where the DIT measures the potentially ancestors affecting this class.

![Depth of Inheritance Tree Diagram](image)

Figure 8.1.: Depth of Inheritance Tree

The different viewpoints defined in [CK94] for this particular metric relate to the insights gained, that the deeper a particular class is located in the inheritance tree, the greater the number of methods inherited are. The latter is related to the predictability of the class’ behavior. The deeper the class is located in the inheritance tree, the more complex it is. Another insight described by [CK94] is based on the
potential reuse of inherited methods which highly depends on the classes location in the inheritance tree.

8.2.2.3. Number of Children

The *Number of Children (NOC)* is the third metric in the OO suite. The NOC is defined by the number of direct subclasses in the classes hierarchy and represents a metric for the *scope of properties*. Relating to the different viewpoints described by [CK94] this metric might also be interpreted as a metric for the reuse of a particular class, since inheritance is a form of reuse. A second viewpoint described by [CK94] is given by the increased likelihood of improper abstractions if a particular class has a high number of children. In this case, it can be assumed that a class with a high number of children might misuse subclassing. The last important viewpoint described, is related to the potential influence of a class on the design that might depend on the number of its children, where classes with a high number of children may require a more rigorous testing of their encapsulated methods.

8.2.2.4. Coupling between object classes

The *Coupling between object classes (CBO)* counts the number of other classes a particular class is coupled with. Coupling in [CK94] is based on the notion of two objects being coupled if one acts on the other. An object is defined to act on another when it uses its methods or instance variables. Chidamber and Kemerer identified that the CBO has an impact on the possibility of reuse, which is dependent on the number of couplings a particular object has. The less objects it is coupled with, the higher its potential level of reuse. Additionally, Chidamber and Kemerer, highlighted that the higher the encapsulation of a particular class is, the less it is sensitive to changes in other parts of the design. In other words, the higher the coupling of a single class is, the higher is the effort to maintain a particular class in a system. But, the CBO has also an influence on the testing effort of a system which for a highly coupled objects needs to be more rigorous.
8. Software Metrics-Related Work

8.2.2.5. Response For a Class

The Response For a Class (RFC) is defined as

\[ RFC = |RS|, \]

where \( RS \) is defined as the response set for the particular class. The response set in [CK94] is defined as the set of methods that potentially can be executed in response to a message received by an object of that class. Chidamber and Kemerer realized that a high RFC results in an increased effort for a tester, where a particular tester needs a wider knowledge of a particular class and the system in advance in order to successfully determine useful test cases. Furthermore, the RFC has an impact on the complexity of a particular class, where a large number of invokable methods indicates an increased level of complexity.

8.2.2.6. Lack of Cohesion in Methods

The Lack of Cohesion in Methods (LCOM) is the sixth metric of the object oriented metric suite. It is used to describe the degree of similarity of methods. The similarity between two methods \( M_1 \) and \( M_2 \) of a class \( C_1 \) is defined as

\[ \sigma = |I_1 \cap I_2|, \]

with \( I_1 \) and \( I_2 \) being the set of instance variables used by the methods \( M_1 \) and \( M_2 \). The LCOM is then build as the sum of the the method pairs where \( \sigma \) is 0, minus the number of methods. The most important interpretation of this metric, is the level of cohesion of a single class. The higher the value of LCOM is, the higher is its cohesion [CK94].

8.2.3. The MOOD suite

After the OO metric suite the Metrics for Object Oriented Design (MOOD) can be thought of as being the second metric suite addressing the object oriented programming/design being of major importance. In contrast to the OO metric suite that has its focus on indicating particular OO concepts of a single class, the MOOD has
its focus on the OO concepts of an application or system. In 1994 Brito e Abreu et al. introduced the first version of the MOOD suite that consisted of eight metrics [BC94]. In [Bri95] the metric suite was revised and two of the initial metrics - clustering and reuse factors - were omitted. In the following we focus on the second version of the metric suite and give an overview of its underlying metrics.

The six remaining metrics are used to interpret the underlying structural concepts of the object oriented paradigm. The Method Hiding Factor (MHF) and Attribute Hiding Factor (AHF) address the concept of encapsulation, where the Method Inheritance Factor (MIF) and Attribute Inheritance Factor (AIF) are used to interpret the concept of inheritance. Finally, Polymorphism Factor (PF) is used to interpret the concept of polymorphism and Coupling Factor (CF) to interpret the concept of message passing [Bri95].

8.2.3.1. Method Hiding Factor

We start our consideration of the metrics the MOOD consist of with the MHF. Like the other metrics of this metric suite, the MHF is based on several direct and indirect metrics:

\( TC \) indicates the total number of classes in the system/application under consideration.

\( M_h(C_i) \) indicates the number of hidden (private and protected) methods.

\( M_v(C_i) \) indicates the number of visible (public) methods.

\( M_d(C_i) \) (the defined methods) is calculated as the sum of the two direct metrics \( M_h(C_i) \) and \( M_v(C_i) \).

In [Bri95] the concept of methods is interpreted towards the structural elements of C++. This means that constructors, destructors, function members, and operator definitions are interpreted as methods, as well. Further, methods and attributes are defined to be hidden, if they have a private or protected visibility, where they are defined to be visible if they have a public visibility. Based on these metrics the MHF is then defined as:
8. Software Metrics-Related Work

\[ MHF = \frac{\sum_{i=1}^{TC} M_h(C_i)}{\sum_{i=1}^{TC} M_d(C_i)} \]

### 8.2.3.2. Attribute Hiding Factor

Like the MHF, the AHF is used to indicate the level of encapsulation of the underlying class. But in contrast, to the MHF, the AHF has its focus on the attributes rather than methods. Therefore, the three attribute based metrics - hidden attributes \( A_h(C_i) \), the visible attributes \( A_v(C_i) \) and the defined attributes \( A_d(C_i) \) - are introduced analogously, where the AHF is defined on \( A_h(C_i) \) and \( A_v(C_i) \):

\[ AHF = \frac{\sum_{i=1}^{TC} A_h(C_i)}{\sum_{i=1}^{TC} A_d(C_i)} \]

### 8.2.3.3. Method Inheritance Factor

Like the AIF, the MIF indicates the level of inheritance of the classes being considered. A set of direct and indirect metrics addressing the inheritance of methods are introduced:

- \( M_i(C_i) \) represents the inherited methods defined in the class \( C_i \), where
- \( M_o(C_i) \) is defined as the number of overriding methods defined in the class \( C_i \).
- \( M_n(C_i) \) represents the new methods defined in the class \( C_i \).
- \( M_d(C_i) \) is used to highlight the sum of the new and overwritten methods \( (M_n(C_i) + M_o(C_i)) \).
- \( M_a(C_i) \) represents the available methods in a single class \( C_i (M_d(C_i) + M_i(C_i)) \).

The MIF indicates the level of inheritance of an application/system and is based on the metrics \( M_i(C_i) \) and \( M_d(C_i) \) of the application’s/system’s underlying classes:
8.2. Metric Suites

\[ MIF = \frac{\sum_{i=1}^{TC} M_i(C_i)}{\sum_{i=1}^{TC} M_a(C_i)} \]

### 8.2.3.4. Attribute Inheritance Factor

The AIF is the second metric addressing the application’s/system’s level of inheritance, where the AIF has its focus on giving an indication of the inheritance based on the attributes in the application’s/system’s underlying classes. Basically, the AIF is defined on several direct and indirect metrics addressing the inherited and overwritten attributes - \( A_i(C_i) \) and \( A_o(C_i) \) - in the application’s/system’s underlying classes. Further, the metrics

\( A_o(C_i) \) representing the number of newly defined attributes in class \( C_i \),

\( A_d(C_i) \) highlighting the number of the newly and overwritten attributes \( (A_n(C_i) + A_o(C_i)) \), and finally

\( A_a(C_i) \) representing the number of available attributes in a single class \( C_i \) \( (A_d(C_i) + A_i(C_i)) \)

are introduced. The AIF is then defined as:

\[ AIF = \frac{\sum_{i=1}^{TC} A_i(C_i)}{\sum_{i=1}^{TC} A_o(C_i)}\]

### 8.2.3.5. Polymorphism Factor

After introducing metrics to address the level of encapsulation and the inheritance of an application/system, the PF addresses the concept of polymorphism. The PF is based on three underlying metrics of which the \( M_o(C_i) \) and the \( M_n(C_i) \) are equally to those defined for the MIF. Further, \( DC(C_i) \) is introduced addressing the number of derived classes. Based on these metrics the PF is defined as:
8. Software Metrics-Related Work

\[
PF = \frac{\sum_{i=1}^{TC} M_o(C_i)}{\sum_{i=1}^{TC} [M_o(C_i) \times DC(C_i)]}
\]

The nominator of the PF’s equation can be interpreted as defining the “number of possible different polymorphism situations”, where a message is either statically or dynamically bound. The number of possible situations can have as many shapes as many times a particular method is overridden. The denominator instead can be interpreted as defining the “maximum number of possible distinct polymorphism situations”. In other words, it represents the case where all of the methods of a particular class \( C_c \) are overwritten by all of its derived classes [Bri95].

8.2.3.6. Coupling Factor

The last metric defined in the MOOD metric suite is the CF. The CF is defined as of being an indicator for the level of coupling. Brito e Abreu et al. defined the CF as:

\[
CF = \frac{\sum_{i=1}^{TC} \sum_{j=1}^{TC} is\_client(C_i, C_j)}{TC^2 - TC - 2 \times \sum_{j=1}^{TC} DC(C_j)}
\] (8.1)

CF is based on the following terms, where

- \( TC^2 - TC \) represents the maximum number of couplings for a system with \( TC \) classes, where
- \( \sum_{j=1}^{TC} DC(C_j) \) describes the maximum number of couplings due to inheritance, and
- \( is\_client \) accepts the values

\[
\begin{align*}
1 & \quad \text{if } C_c \Rightarrow C_s \land C_c \neq C_s \Rightarrow (C_c \rightarrow C_s) \\
0 & \quad \text{otherwise.}
\end{align*}
\]

The latter term represents the client/supplier relation. This means that a relation exist if the client class \( C_c \) contains at least one reference to a method or attribute of the supplying class \( C_s \). \( (C_c \rightarrow C_s) \), in turn, is used to indicate the inheritance
relation for class $C_c$ inheriting from class $C_s$. As the CF is defined to evaluate the non-inheritance, coupling to particular classes $C_s$ and $C_c$ should not be tied by any inheritance relationship [Bri95].

From a more abstract perspective the nominator can be interpreted as a representative for the number of couplings not imputable to inheritance. The denominator instead can be interpreted as of being the maximum number of non-inheritance couplings in a system comprised of $TC$ classes [Bri95].

### 8.3. Design Measures/Metrics

In the previous section we highlighted the metric suites that in the literature are denoted as of being of highest importance. In contrast to the Halstead metric suite presented at the beginning of the previous section, the OO metric suite and the MOOD suite have their focus on highlighting and measuring the concepts of the object oriented paradigm. The OO considers the concepts of the object oriented paradigm from a localized, class centric point of view, where the underlying metrics of the MOOD suite are defined from the application’s/system’s point of view.

However, the Halstead, the OO metric suite and the MOOD suite are based on a source code representation of an application/system. The Software Design (SD) Metric Suite is a collection of a large number of well known and individual design metrics found in recent literature, such as [LK94, BDM97, LLWW95, LC94, MGP03]. The majority of the metrics in the SD metric suite are direct metrics that are used to calculate the number of underlying elements of a particular concept like the number of operations or the number of classes and so on.

Basically, the metrics of the SD metric suite are categorized according to the basic elements and diagram types of the UML. Each category - class, interface, package, use case, state machine, activity, component, node and diagram measures/metrics - consists of at least seven metrics. For each of these basic categories several sub-categories are defined that address concepts such as the size, the level of coupling or the complexity of the respective category. Each of the sub-categories in turn is comprised of at least one up to eight metrics indicating the level of a sub-category based on different point of views.
A condensed overview of the metrics the SD metric suite is comprised of and its corresponding categorization is given in [Chr09]. As the presented metrics are mainly direct that count a particular issue, we avoid to go into further detail on these metrics and refer to [Chr09] for an overview of these.

8.4. Component Measures/Metrics

Metrics addressing software components can basically be distinguished in those addressing the local, internal issues of a software component and those addressing the issues related to the component’s underlying composition hierarchy. In recent publications mainly metrics are introduced that consider the internal concepts of a software component in an isolated way.

8.4.1. Cho, Kim and Kim

The component metric introduced by Cho, Kim and Kim [CKK01] are based on flat component models such as COM+, EJBs and CCM. The metrics introduced basically co a software component in an isolated way, where the central distinction of hierarchical components - acquired and contained components - that are fundamental in hierarchical component models, is not taken into regard. The authors introduce metrics that are used to indicate three central concepts complexity, customizability and reliability.

8.4.1.1. Complexity

Cho, Kim and Kim identified that the cyclomatic complexity can not be used to determine the complexity of a single component because of inheritance in a component, but that it can be combined with other metrics to determine the complexity of a component [CKK01, p. 421]. As a metric to determine the component's complexity the Component Complexity Metric (CCM) is introduced. The CCM is based on four metrics addressing different issues concerned with the complexity of a class' underlying elements.
8.4. Component Measures/Metrics

Component Plain Complexity

The Component Plain Complexity (CPC) is used to determine the complexity of the component itself, where it calculates the sum of classes, abstract classes, interfaces - $CmpC$ - and the complexity of classes - $\sum_{i=1}^{m} CC_i$ - and methods - $\sum_{j=1}^{m} MC_j$ - [CKK01, p. 421].

$$CPC(C) = CmpC + \sum_{i=1}^{m} CC_i + \sum_{j=1}^{m} MC_j$$

For the calculation of the $CmpC$ weighting factors are introduced to weight the basic elements (except interfaces) that are contained in a component (classes and methods). Cho, Kim and Kim identified that two different kinds of classes are contained in a single component - external and internal classes. They define external classes to be classes that are imported from reused libraries or packages, where internal classes are defined to be classes that result from the component analysis and design.

$$CmpC = \sum_{i=1}^{m} (\text{Count}(C_i) \times W(C_i)) + \sum_{j=1}^{m} (\text{Count}(I_j)) + \sum_{k=1}^{o} (\text{Count}(M_k) \times W(M_k))$$

The complexity of a class that is contained in a component is calculated as the sum of the class’ attributes, where they distinguish between single attributes (SA) and complex attributes (CA), e.g. which type is a class.

$$CC = \sum_{i=1}^{m} (\text{Count}(SA_i) \times W(C_i)) + \sum_{j=1}^{m} (\text{Count}(CA_j) \times W(CA_j))$$

The complexity of a method is based on the parameters of the method, where a distinction between simple and complex attributes is made as well - simple parameter (WP) and complex parameter (CP). The CP is weighted as well as it usually contains other arguments that increase the method complexity [CKK01, p. 421].

$$MC = \sum_{i=1}^{m} (\text{Count}(SP_i) \times W(C_i)) + \sum_{j=1}^{m} (\text{Count}(CP_j) \times W(CP_j))$$
Component Static Complexity

The Component Static Complexity (CSC) is used to measure the complexity of a component’s internal structure with a static view. The static complexity is defined as the sum of the weighted - \( W(R_i) \) - relationships amongst classes - \( Count(R_i) \) - in a component. Weights are used to distinguish between the different kind of relationships - Dependency, Aggregation, Generalization and Composition - between a component’s classes.

\[
CSC = \sum_{i=1}^{m} (Count(R_i) \times W(R_i))
\]

Component Dynamic Complexity

In contrast to the CSC that has its focus on the internal structure of the component, the Component Dynamic Complexity (CDC) is used to determine the dynamic complexity based on how many messages are passed in a component.

\[
CDC = \sum_{i=1}^{m} DC(IM_i)
\]

The complexity of an interface method - \( DC(IM) \) - is based on the number of messages passed \( \text{Count}(Msg_i) \), on the frequency of the messages passed \( \text{Freq}(Msg_i) \) and the the complexity of each message \( \text{MC}(Msg_i) \).

\[
DC(IM) = \sum_{i=1}^{m} \text{Count}(Msg_i) \times \text{Freq}(Msg_i) \times \text{MC}(Msg_i)
\]

Component Cyclomatic Complexity

After introducing three metrics addressing the design time complexity of a component, the Component Cyclomatic Complexity (CCC) is used to determine the component’s complexity based on its realization in terms of a source code representation of the component.
8.4. Component Measures/Metrics

\[ CCC = CmpC + \sum_{i=1}^{m} CC_i + \sum_{j=1}^{m} MC_j + \sum_{k=1}^{o} CCM_k \]

While the complexity of methods - \( MC_j \) - and classes - \( CC_j \) - is equal to the ones already used for CPC, the cyclomatic complexity for methods CCM is based on the source code representation where McCabe’s cyclomatic complexity is used to define the CCM:

\[ \sum_{k=1}^{o} CCM_k = \text{edges} - \text{nodes} + 2 \]

8.4.1.2. Customizability

Cho, Kim and Kim defined customizability to be one of the central characteristics of component based development. To measure a component’s customizability the metric component variability (CV) is introduced:

\[ CV = \frac{\sum_{i=1}^{m} \text{Count}(CVM_i)}{\sum_{j=1}^{n} \text{Count}(CIM_j)} \]

CV is comprised of two other metrics. The \( \text{Count}(CVM) \) addresses the number of methods for customization and \( \text{Count}(CIM) \) addresses the number of methods declared in each interface. The \( CVM_i \), in turn is based on the metrics \( \text{Count}(CVM_a) \) that is used to determine the number of method for attribute customization, \( \text{Count}(CVM_m) \) that is defined to determine the number of methods for behavior customization and finally the \( \text{Count}(CVM_w) \) addressing the number of methods for workflow customization [CKK01, p. 423]. Weighting factors are further introduced to weight the behavior and workflow customization as these are thought as of being more complex than attribute customization.

\[ CVM = \sum_{i=1}^{m} \text{Count}(CVM_a_i) + \sum_{j=1}^{n} \text{Count}(CVM_m_j) + \sum_{k=1}^{o} \text{Count}(CVM_w_k) \]
8. Software Metrics-Related Work

8.4.1.3. Reusability

Basically two different kinds of reusability metrics are introduced by Cho, Kim and Kim. The first one - CR - addresses the reusability of the component itself, where the second one has its focus on the reusability of a component based on how it is reused in a particular application [CKK01, p. 423].

The component itself reusability (CR) may be applied at the design phase, where it is calculated as fraction of the sum of $\text{Count}(CCM)$ and the sum of $\text{Count}(CIM)$:

$$CR = \frac{\sum_{i=1}^{m} \text{Count}(CCM_i)}{\sum_{j=1}^{n} \text{Count}(CIM_j)}$$

The $\text{Count}(CCM)$ is defined as the sum of each interface method for providing common functions among several applications in the domain [CKK01, p. 423]. The $\text{Count}(CIM)$, instead, is defined as the number of interface methods defined for a component.

The second reusability metric is concerned with the reuse level of a component in a particular application. The Component Reuse Level (CRL) is further refined and two different kinds of CRL metrics are introduced. The first one - the $CRL_{LOC}$ - is used to determine the reuse level on basis of the LOC. The $CRL_{LOC}$ is based on the $\text{Reuse}(C)$ that is used to determine the lines of code of the reused component in an application, where the $\text{Size}(C)$ is defined as the total number of lines of code. The $CRL_{LOC}$ then is used to give the LOC based reuse level as:

$$CRL_{LOC} = \frac{\text{Reuse}(C)}{\text{Size}(C)} \times 100\%$$

The $CRL_{LOC}$ is based on the functionality the component supports divided by the required functionality in an application.
8.4. Component Measures/Metrics

8.4.1.4. Conclusion

Cho, Kim and Kim introduce sound definitions of software metrics to determine a component's complexity, customizability and reusability. However, they do not address the criticism of the complexity metrics proposed by Weyuker [Wey88].

The CDC is concerned with the messages passed between the classes in a component. Cho, Kim and Kim do not further define which complexity metric should be applied to measure the CDC. However, as it only regards the methods in an isolated way (summing up the complexity of the methods) the increased complexity based on the concatenation of methods is neglected.

The CCM defines the complexity of a particular method based on the cyclomatic complexity introduced by McCabe [Mcc76]. As the complexity of each method is used in an isolated way as basis for the calculation of the CCC, properties like the missing additivity of $\vartheta(G)$ is neglectable at this point.

So far our criticism is concerned with properties related to measurement theory. However, the major criticism on the metric suite of Cho, Kim and Kim is based on the fact that they do not further define what a component is. Intuitively they assume that a component internally needs to be object oriented. due to the analysis of common component definitions and component models this does not necessarily need to be the case. This, however, limits the applicability of the metric suite of Cho, Kim and Kim.

8.4.2. Washizaki, Yamamoto and Fukazawa

In contrast to Cho, Kim and Kim that do not further define on what they think of a component is, Washizaki, Yamamoto and Fukazawa [WYF03] go into detail on this issue before reusability metrics for components are introduced. Like Cho, Kim and Kim they distinguish between two activities: development of components for reuse and development of component based systems by reusing components.

A reusability model for black box components is introduced based on McCall’s Factor-Criteria-Metrics (FCM) defined in [MRW77]. The FCM is used as foundation of the ISO/IEC 9126, where it is used to assess the quality of a software entity.
In their reusability model reusability is decomposed in three factors: understandability, adaptability and portability.

Further, four criteria are introduced to determine these factors. Criteria, generally are used to describe the quality factors, where from an assessment points of view these are determined as of being design level attributes. Metrics instead are defined to be product level attributes and are used to measure the criteria.

The existence of meta-information and the observability are used to describe a software components understandability at the design level. The customizability, instead, is used to describe the adaptability and finally the external dependency is used to define the portability on the design level. The metrics introduced by Washizaki, Yamamoto and Fukazawa are used to measure the criteria defined on the design level.

According to their model five metrics are introduced to measure the criteria defined in their reusability model. Although they used a general definition of a software component the metrics introduced have been defined on basis of the information available for JavaBeans.

### 8.4.2.1. Existence of meta-information

The existence of meta-information (EMI) is defined as: \( EMI(C) \) depends on whether the BeanInfo class corresponding to the target component \( c \) is provided [WYF03, p. 4]:

\[
EMI(c) = \begin{cases} 
1, & \text{BeanInfo class exist} \\
0, & \text{otherwise}
\end{cases}
\]

If \( EMI(c) \) is 1 it is expected that users of \( c \) can easily understand the usage of \( c \) which is assumed by the developer of \( c \).

### 8.4.2.2. Rate of Component Observability

The Rate of Component Observability (RCO) is used to indicate the degree of observability for users of the component, where the observability should be high to
8.4. Component Measures/Metrics

understand the behavior of the component form outside the component. If the observability is too high, instead, it might be difficult to find the important readable properties in the set of readable properties [WYF03, p. 5]. The RCO is based on two direct metrics. The $P_r(c)$ is defined as the number of readable properties in $c$, where $A(c)$ is defined as the number of fields in $c$’s facade class.

$$RCO(c) = \begin{cases} \frac{P_r(c)}{A(c)}, & A(c) > 0 \\ 0, & otherwise \end{cases}$$

### 8.4.2.3. Rate of Component Customizability

Similar to the RCO indicating the degree of observability, the Rate of Component Customizability (RCC) is used to indicate the degree of a component’s customizability for users of $c$. In order to easily adapt the settings of the component to the user’s requirements the $RCC(c)$ should be high. However, if the $RCC(c)$ is too high, the high customizability violates the encapsulation of the component. Like the RCO, the RCC is based on two direct metrics, where the $A(c)$ is equal to the one defined for RCO. The $P_w(c)$ is defined as the number of writable properties.

$$RCC(c) = \begin{cases} \frac{P_w(c)}{A(c)}, & A(c) > 0 \\ 0, & otherwise \end{cases}$$

### 8.4.2.4. Self-Completeness of Component’s Return Value

The Self-Completeness of Component’s Return Value (SCCr) is defined as the ratio between the number of business methods without any return return value ($B_V$) and the number of business methods ($B$).

$$SCCr(c) = \begin{cases} \frac{B_V(c)}{B(c)}, & B(c) > 0 \\ 0, & otherwise \end{cases}$$

Washizaki, Yamamoto and Fukazawa assume that the SCCr can be used to determine the self-completeness of a component and therefore its degree of dependencies to external elements. Although they state that the smaller the number of business
methods without return value is, the smaller is the possibility that the component has dependencies. This is obviously wrong. It would be expectable that the higher the number of business methods without return value is, the smaller the possibility is that the component has external dependencies. The latter, however, is also underpinned by the confidence interval defined for the SCCr being between 0.61 and 1.0 [WYF03, p. 5].

8.4.2.5. Self-Completeness of Component’s Parameter

Like the SCCr, the Self-Completeness of Component’s Parameter (SCCp) is based on the business methods of a component. However, the SCCp has its focus on the parameters passed with the business methods. The term $B_c$ is used to determine the number of business methods without any parameters. Washizaki, Yamamoto and Fukazawa define the ratio between $B_c$ and $B$ as the self-completeness of a component based on its parameters.

$$SCCp(c) = \begin{cases} \frac{B_p(c)}{B(c)}, & B(c) > 0 \\ 0, & otherwise \end{cases}$$

Like in the definition of the SCCr Washizaki, Yamamoto and Fukazawa state that the smaller the number of business methods without parameters are, the smaller the possibility of having a dependency outside the component [WYF03, p. 6]. This kind of relationship between the number of business methods without parameters and the number of business methods is wrong as well. The relationship between $B_p$ and $B$ is much more a is higher - is smaller relationship that is also underpinned by the confidence interval defined for the SCCp.

8.4.2.6. Conclusion

In contrast to the metrics introduced by Cho, Kim and Kim that have been introduced without the definition of an underlying model, the metric suite introduced by Washizaki, Yamamoto and Fukazawa clearly defines the relationship between the criteria, factors and metrics their metric suite is based on. Further they define the underlying model of JavaBeans as the component model their metric suite is defined on.
Therefore the metric suite is tailored to the issues related to JavaBeans. However, most of the metrics can be generalized towards the model introduced in chapter 5 HIERARCHICAL COMPONENT MODEL. Anyway, we basically disagree with the authors of [WYF03] interpretation of some of their metrics.

For instance, we disagree with the interpretation of the SCCr and the SCCp as of being indicators for the degree of dependencies to elements external to the component. Basically, we think that it is not correct to treat each type of return values and parameters in the same way. As they use JavaBeans as underlying model for the definition of the metric suite we would at least expect that they would make a difference between primitive and complex types such as classes. We further would expect that a difference is made between the types defined in Java and those defined elsewhere. The ones defined in Java are a fundamental part of the underlying model and in our opinion should have a smaller influence on the external dependency than classes or other elements that are not part of the underlying model.

### 8.4.3. Narasimhan and Hendradaja

In contrast to the metric suite introduced by Washizaki, Yamamoto and Fukazawa presented in the previous section that is tailored to the notion and underlying elements of JavaBeans, the metrics introduced by Narasimhan and Hendradaja are based on a more general notion of software components. As they focus on the composition of the component itself and its integration in an application [NH04, p. 3] a Component Integration Description Language (CIDL) is defined, which is used as underlying model of their software component metrics.

#### 8.4.3.1. Component Integration Description Language

They define a software application to consist of modules, classes and operations which include procedural codes, variables and components [NH04, p. 3]:

\[
\text{<Software System>} ::= \text{<modules> | <classes> | <operations> | <variables> | <component>}
\]

\[
\text{<Component>} ::= \text{<modules> | <classes> | <operations> | <variables> | <component>}
\]
Several sub types of components are defined, that distinguish components based on their content. An operation-component is defined to consist only of operations or traditional procedures. A class-component, instead, is defined to consist of classes, where a module-component is defined as consisting only modules. Finally the term super-component is used to describe components that are a combination of components, operations, variables and classes.

8.4.3.2. Component Packing Density

Based on the CIDL two metrics addressing the complexity of components and component based software systems are introduced. The Component Packing Density (CPD) gives an indication of the component’s/software system’s complexity based on the ratio of the number of a particular constituents and the number of components:

$$CPD_{\text{constituent\_type}} = \frac{\# < \text{constituent} >}{\# \text{component}}$$

The constituents of the CPD can be one or more of LOC, number of objects/classes, number of modules or number of operations. Narasimhan and Hendradaja assume that the CPD metric can be used to show the density of components interacting with each other. They further assume that a higher density also means a higher complexity. Finally, they assume that the CPD can be used to predict the kind of underlying type of application, where the CPD for business applications basically should be different from the CPD of real time applications, for instance.

8.4.3.3. Component Interaction Density

The second metric that is defined in [NH04] is the Component Interaction Density (CID). Narasimhan and Hendradaja define that an interaction among components is based on the use of the component’s interface or through the consumption of the component’s publicly visible events [NH04, p. 4].

They recognized that the points of interaction (interfaces or events) are closely related to the reliability of a component. They further realized that a higher degree
of interactions causes a higher complexity of the component. To measure the interaction density amongst components the Average Interaction Density (AID) is introduced, that in turn is based on three additional metrics: the Interaction Density of a Component (IDC), the Incoming Interaction Density of a Component (IIDC) and finally the Outgoing Interaction Density of a Component (OIDC).

The IDC is defined as the ratio between the interactions actually used in a systems and the number of interactions available in the component [NH04, p. 5]:

\[ IDC = \frac{\#I}{\#Imax} \]

The IDC is further refined and the same basic idea is used to define the IIDC and the OIDC. Incoming interactions are either received interfaces that are required by the component or the received events of a component:

\[ IIDC = \frac{\#I_{IN}}{\#Imax_{IN}} \]

The OIDC, instead, is based on the component’s number of provided interfaces used and the component’s number of events consumed in the software system:

\[ OIDC = \frac{\#I_{OUT}}{\#Imax_{OUT}} \]

The AID is then defined as the sum of the interaction density of each component in a software system divided by its number of components [NH04, p. 5]:

\[ ADC = \frac{IDC_1 + IDC_2 + \ldots IDC_n}{\#components} \]

In contrast to the IDC, IIDC and OIDC that are used to determine the particular interaction densities within a particular component, the AID is used to determine the interaction density of a software system. Narasimhan and Hendradaja assume that a low level of the interaction density indicates a simple system, where a high value indicates a complex systems.
8. Software Metrics-Related Work

8.4.3.4. Conclusion

The metrics introduced by Narasimhan and Hendradaja basically have their focus on a component and the system it is integrated in. To cover the issues related to software components and their encapsulating software system a clearly defined model, the metrics are based on, is introduced.

Two metrics addressing the complexity of a component based software system are introduced. The CPD, for instance, is used to determine the component’s complexity based on the ratio between the particular constituents of the software system and the number of components. Further interaction density metrics are introduced that have their focus on the interactions between the component under consideration and and other parts of the systems. Basically, these can be distinguished in those addressing the different kinds of interaction of a component - IDC, IIDC and OIDC - and those addressing the interactions between components in a software system - ADC.

8.5. Summary

In this chapter we gave an overview of common metrics addressing particular aspects of software entities. We started this chapter by giving an overview of metrics that are related to the complexity of software entities. These were one of the most used metrics in practice, the LOC, and the cyclomatic complexity.

We further gave an overview of the most important metric suites discussed in the literature. We started with the Halstead suite that is based on textual based representations, where in practice it is historically used to determine aspects of procedural languages. In contrast, the metrics encapsulated in the metric suite of Chidamber and Kemerer - the object oriented metric suite - and the MOOD metric suite of Brito e Abreu et al. address the issues of object orientation.

Metrics related to the model/design of software applications are discussed in the SD metric suite of Wüst. The software design metric suite is a collection of well known metric defined in the literature.

Finally, we presented metrics related to software components. The most important ones are the metric suite defined by Cho, Kim and Kim that have their focus on
component related metrics based on complexity. The metric suite of Washizaki, Yamamoto and Fukazawa, instead, address issues related to flat component models, in detail JavaBeans. Finally, we presented the component metrics of Narasimhan and Hendradaja that have their focus on measuring the complexity of a component based system.
In the previous chapters we presented the foundations for the development of metric suites for hierarchical component based systems. We defined a hierarchical component model based on the impressions and definitions of commonly used hierarchical component models in chapter 5 HIERARCHICAL COMPONENT MODEL. Further, we discussed the theoretical properties of metrics in the context of measurement theory resulting in a method for the analysis of the underlying properties of a metric.

Generally, metrics are defined for a particular context. The Halstead metric suite, for example, was developed for text-based documents. Anyway, these metrics are often used in a different context without ensuring that the metrics fulfill the required theoretical properties of the given context. In this chapter we focus on the properties software metrics need to fulfill to be hierarchy-aware.

Due to the importance of operations in the context of component based systems, we focus on the metrics with respect to a particular operation - the invocation-chain metrics. Operations in component based system mostly fulfill their contractually specified obligations with the help of other operations, which implies that an operation calls other operations. These operations may be encapsulated in the component under consideration, in components that are i-contained in the component
9. *Hierarchy-aware Measurement*

under consideration or in components that are i-acquired by the component under consideration.

Based on where the operations of the invocation-chain are located at, e.g. the component under consideration, we introduce a naming scheme for hierarchy-aware measurement. The resulting categories are discussed in great detail afterwards. Although other metrics than the metrics that are defined with respect to a particular operation exist, we neglect these in this thesis and defer these to future research.

For our consideration of the hierarchy-aware categories we use a simple metric the *Number of Statements* (NOS). We discuss the properties of each category on behalf of the respective category-based NOS, where we first give an *Object Constraint Language* (OCL)-based description of the metric. Afterwards, we discuss the theoretical properties a category-based metric needs to fulfill. Finally, we relate the category-based metric with the quality characteristics, where we show the influence of the category on the indication.

The category-based consideration has a major impact on some of the quality characteristics such as reliability. Anyway, metrics like the NOS give an absolute value of an aspect that highly depends on the underlying context. Therefore, we introduce a relative indication in terms of the delegation ratios. Suitable indications based on delegation ratios are the ratio between the functionality covered by a local operation and the functionality covered by the complete invocation-chain.

These are discussed at the end of this chapter. Comparing two components covering the same functionality delegation ratios allow a powerful judgment on which one to choose. A component delegating most of its functionality to operations of i-acquired components in our view is less reliable than a component delegating most of its functionality to operations encapsulated in the component under consideration or its i-contained components.

### 9.1. Categorization

Due to the importance of operations and the invocation-chains in the context of component based development we introduce an operation-centric naming scheme
for hierarchical component based metrics in this section. Basically it is a component’s operations that provide the functionality of the component. The simplest category - the *Local Operation Metrics* category - considers an operation $op$ in an isolated way. The invocation-chain that results from following up each method call of the affected operations when $op$ is invoked is completely ignored in this category.

The second category only takes the method calls to the operations encapsulated in the component under consideration into account. Not following up the invocation-chain to operations of other components, the invocation-chain is considered in a shallow, component-centric way. Thus, we define the second category as the *Shallow Operation Metrics* category.

If we follow up the invocation-chain to the operations of other components, we basically consider the invocation-chain in deep. Thus, the last category is called *Deep Operation Metrics*. Anyway, the *Deep Operation Metrics* category is further refined, as based on the different association types that are found in the composition hierarchy we distinguish between method calls to operations that are provided by components being i-acquired or i-contained by the component under consideration. Thus, we define three sub-categories of the *Deep Operation Metrics*.

The sub-category is the *Deep-Contained Operation Metrics* category. The metrics of this category have their focus on issues such as complexity that are covered by the operations provided by the component under consideration and its directly or indirectly i-contained components which are affected by the invocation-chain. The second one is the *Deep-Acquired Operation Metrics* category. In this category only the issues that are covered by the operations provided by the i-acquired components which are affected by the invocation-chain are taken into account. Finally, the third sub-category does not distinguish between operations provided by the component under consideration, its i-contained or its i-acquired components. We call this category the *Deep-Complete Operation Metrics*.

Based on the categories that are defined for operation-centric composition hierarchy metrics we introduce a naming scheme for the hierarchy-aware category-based metrics. A *Deep-Contained Operation Metrics* category-based metric like the cyclomatic complexity is called the *Deep-Contained Cyclomatic Complexity*. A *Shallow Operation Metrics* category-based cyclomatic complexity is then called the *Shallow Cyclomatic Complexity* and so on. In table 9.1 the naming scheme for the category-
9. Hierarchy-aware Measurement

based cyclomatic complexity $\vartheta(G)$ and the category-based Number Of Statements (NOS) is presented. For other hierarchy-aware metrics the naming scheme is applied analogously.

<table>
<thead>
<tr>
<th></th>
<th>Local</th>
<th>Shallow</th>
<th>Deep</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contained</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Acquired</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Complete</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Cyclomatic complexity $\vartheta(G)$  
$\vartheta(G)$  
Deep-Contained $\vartheta(G)$  
Deep-Acquired $\vartheta(G)$  
Deep-Complete $\vartheta(G)$

Number of Statements NOS  
NOS  
Deep-Contained NOS  
Deep-Acquired NOS  
Deep-Complete NOS

Table 9.1.: Naming Scheme for hierarchy-aware category-based metrics

9.2. Local Operation Metrics

We start our consideration on hierarchy-aware-measurement with the Local Operation Metrics. In this context we consider operations in an isolated way. Dependencies to other elements that are outside the operation, e.g. other operations or components of the composition hierarchy are not considered.

To clearly define metrics in this context we enhance the component model that was introduced in chapter 5 Hierarchical Component Model by adding a new class - LocalOperationMetrics - that is used to keep track of all local operation metrics (see figure 9.1).

9.2.1. OCL-based Definition

Starting from the LocalOperationMetrics-element we can clearly define local operation metrics based on the hierarchical component model. For a better understanding we use a simple example - the Number of Statements (NOS) - for which the theoretical analysis is performed in detail. However, the discussion is not particular to the NOS and is analogously applied to other local operation metrics. According to the naming scheme introduced in the previous section these are e.g. the
9.2. Local Operation Metrics

Local Number of Branches, the Local Number of Operators, the Local Number of Operands and so on.

Using OCL we define the Local Number of Statements in an operation as:

```ocl
context Body
def nestedNOS(body:Body):Integer = body.Statement.select(oclIsTypeOf(SimpleStatement))->size() + body.Statement.select(oclIsTypeOf(ComponentStatement))->iterate(cs : ComponentStatement; result: Integer = 0 | result + nestedNOS(cs.body))

context LocalOperationMetrics
```

In the OCL-based definition of the Local-NOS, we first introduced a helper method - the nestedNOS - in the context of Body. The nestedNOS recursively calculates the number of statements of all bodies within the body that is passed as parameter. Further, the Local-NOS is defined in the context of LocalOperationMetrics where it is added as an attribute. The attribute defines a query to the body of an operation and then applies the nestedNOS to determine directly the number of all statements in the operation’s body.
9. Hierarchy-aware Measurement

9.2.2. Theoretical Properties

The analysis of the theoretical properties of local operation metrics and of all other categories are based on the methodology introduced at the end of chapter 7 SOFTWARE MEASUREMENT. Although we discuss the theoretical properties of the respective category on behalf of a category-based NOS, other invocation-chain metrics of the respective category are determined analogously.

In the first step, the set $A$ of the underlying relational system $U$ needs to be determined. The set $A$ defined by the crate of apples, that we used as initial example for the theoretical foundations, is based on the individual apples. Analogously, the set $A$ we are dealing here, is based on the operations. Thus, $A$ is defined as $A = \{a_1, a_2, a_3, \ldots, a_n\}$, with $a \in A$ being an individual operation.

Relations between the number of statements, we are basically interested in, are the has more statements than ($R_1$), has more or the same number of statements than ($R_2$) and has the same number of statements as ($R_3$) relationship. Thus, the relational system $U$ is defined as $U = (A, R_1, R_2, R_3)$.

In this paragraph we determine the basic properties of the relations identified as being of interest, with $a, b \in A$:

$R_1$: **Symmetry**: $a$ has more statements than $b$ implies that $b$ can not have more statements than $a$. In other words: $aR_1 b \rightarrow \neg bR_1 a$. Thus, $R_1$ is asymmetric.

**Reflexivity**: The relation $aR_1 a$ does not hold for any $a \in A$. The relation $\neg aR_1 a$ is not true either. $R_1$ is nonreflexive.

**Transitivity**: For $a, b, c \in A$ the relations $a$ has more statements than $b$ and $b$ has more statements than $c$ imply that $a$ has more statements than $c$. This means that $R_1$ is transitive.

$R_2$: **Symmetry**: $a$ has more or the same number of statements than $b$ does not imply that $b$ can not have more statements than $a$, but it implies that if $b$ has more or the same number of statements than $a$ that $a = b$. Thus, $R_2$ is antisymmetric.

**Reflexivity**: The relation $aR_2 a$ holds for any $a \in A$. Thus, $R_2$ is reflexive.

**Transitivity**: For $a, b, c \in A$ the relations $a$ has more or the same number of statements than $b$ and $b$ has more or the same number of statements than $c$ imply that $a$ has more or the same number of statements than $c$. This means that $R_2$ is transitive.
9.2. Local Operation Metrics

c imply that a has more or the same number of statements than c. This means that $R_2$ is transitive.

$R_3$: The discussion of $R_1$ and $R_2$ can be applied analogously for $R_3$. The analysis of the properties of $R_3$ is discarded at this point. The results, however, are presented in table 9.2.

The last property that needs to be considered at this point is the completeness. Therefore, we introduce the relation weakly preferred to (W) referring to the number of statements. We prefer a to b, or b to a except $a = b$. Thus, the relations $R_1, R_2$ and $R_3$ are meant to be complete. An overview of the properties of the relations $R_1, R_2$ and $R_3$ are illustrated in table 9.2.

<table>
<thead>
<tr>
<th></th>
<th>$R_1$</th>
<th>$R_2$</th>
<th>$R_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Symmetry</td>
<td>asymmetric</td>
<td>antisymmetric</td>
<td>symmetric</td>
</tr>
<tr>
<td>Reflexivity</td>
<td>nonreflexive</td>
<td>reflexive</td>
<td>reflexive</td>
</tr>
<tr>
<td>Transitivity</td>
<td>transitive</td>
<td>transitive</td>
<td>transitive</td>
</tr>
<tr>
<td>Completeness</td>
<td>complete</td>
<td>complete</td>
<td>complete</td>
</tr>
</tbody>
</table>

Table 9.2.: Basic properties of the relations has more statements than, has more or the same number of statements than and has the same number of statements than.

As $R_1$ is asymmetric, transitive and complete it is denoted as a strict simple order (see table 7.4). $R_2$, instead, is antisymmetric, reflexive and transitive. Thus $R_2$ is a partial order. Due to the properties of $R_3$, it is defined to be a quasi order.

The steps 5 to 9 of the introduced method have their focus on determining a numerical relational system appropriate for the relational system $U$. Basically, a suitable representation of the number of statements for operations is the set of nonnegative integers including 0 ($\rightarrow \mathbb{N}_0$). Each operation can be mapped to a particular number determining the number of statements.

The relations defined by $U = \{A, R_1, R_2, R_3\}$ are mapped to $> (R_1)$, $\geq (R_2)$ and $= (R_3)$. Thus, the numerical relational system used to determine the number of statements in operations is defined by $\mathfrak{S} = \{\mathbb{N}_0, >, \geq, =\}$. As the analysis of the basic properties is equivalent to the ones discussed for the relational system $U$ we only give an overview of the basic properties of the relations of $\mathfrak{S}$ (see table 9.3). The relations of $U$ and the corresponding relations of $\mathfrak{S}$ have the
same properties. This implies that these have the same kind of order (Step 8 and 9).

Operations on a single component’s method such as concatenation do not need to be taken into account in this category as local operation metrics have their focus on operation-centric concerns. Thus, we define \( \mathcal{U} \) not to have any operations in this category, whereas operations need to be taken into account for the other categories.

As basically the only way to obtain the number of statements of an operation is to count them, the scale \( (\mathcal{U}, \mathcal{R}, f_{\text{Local}}) \) is an absolute scale (Step 13). The last step, step 14, is used to determine whether \( f \) is a measure or a metric. In case of \( f_{\text{Local}} \) the properties of measures are fulfilled.

Anyway, other local operation metrics that depend on the invocation-chain are analyzed analogously.

### 9.2.3. Indicator

Having identified the theoretical properties of a local operation metric, we turn our attention to the indication towards quality characteristics. At this point we focus on metrics that turn out to be similar to the Local-NOS.

The Local-NOS discussed in this section is closely related to some of the metrics presented in the previous chapter. These are the LOC that is applied on the source code representation of a software artifact and the cyclomatic complexity based on a graph based notion of the software artifact. In detail, often the LOC is restricted to counting only the number of lines of code where a statement can be found. As more than a single statement can be found in a single line, the numerical value of LOC will be smaller or equal to the Local-NOS.
In contrast to the cyclomatic complexity, that distinguishes between different kinds of statements (branch and loop statements (predicate nodes) vs other statements), the Local-NOS does not and treats all of them equally. As the cyclomatic complexity is based on the difference between the branch- and loop-statements and other kinds of statements, it is smaller or equal to the Local-NOS.

As both LOC and cyclomatic complexity are used to indicate the complexity of operations in source code representations of software artifacts, the Local-NOS is a suitable indicator for the complexity of an operation considered in an isolated way.

### 9.2.4. Summary

We used the Local-NOS as a representative for local operation metrics. Other local operation metrics, direct and indirect ones, can be obtained in a similar way as performed for the Local-NOS. We applied the proposed methodology to determine the theoretical properties of the Local-NOS in great detail. For other invocation-chain based metrics of the local operation metrics category the analysis is performed analogously.

### 9.3. Shallow Operation Metrics

Like the local operation metrics where all elements in an operation's body are treated in an isolated way, the shallow metrics treat all elements within a component in an isolated way.

**Definition** (Shallow Metrics). Shallow metrics consider the component under consideration in an isolated way. Only the component owning the operation is considered.

For the discussion and definition of shallow metrics we enhance the component model by a single element that is used to collect the metrics regarding an individual component. In figure 9.2 we illustrate an excerpt of the component model that is of importance for the shallow metrics.

Some shallow metrics are defined in a similar way as the Local-NOS in the previous section e.g. the *Number of Operations* and the *Number of Attributes*. These are
component-centric, which means that they are contained in a *shallow component metrics* category. Others, instead, are an enhancement of the local operation metrics. Thus, these are categorized as *shallow operation metrics*. As discussed in the last section local operation metrics treat the elements in an operation in an isolated way. Method calls to other operations are not followed up by these.
9.3. Shallow Operation Metrics

**Definition** (Shallow Operation Metrics). *Shallow metrics consider the component under consideration in an isolated way. Only method invocations to operations that are encapsulated by the component under consideration are is considered.*

Operations are a central concept of components. It is a component’s operations that expose its functionality. Due to the importance of a component’s operations we are mainly interested in how to consider shallow metrics with respect to a particular operation. Like the category of local operation metrics we discuss the category of shallow operation metrics on behalf of a particular metric - the *Shallow Number of Statements*. Metrics of the shallow component category, however, are not in the focus of this thesis and are deferred to future research.

9.3.1. OCL-based Definition

Based on OCL we define the *Shallow Number of Statement* $NOS^d_{Shallow}(op)$ as:

```plaintext
context Body
def nestedNOS-Shallow(comp: Component, body:Body):Integer =
    body.Statement.select(oclIsTypeOf(SimpleStatement))->iterate(cs : SimpleStatement; resultSimple: Integer = 0 |
    if cs.oclIsTypeOf(MethodCall) then
        if comp = cs.Operation.Component then
            resultSimple +
            nestedNOS-Shallow(comp, cs.Operation.Body) + 1
        else resultSimple + 1
        endif
    else resultSimple +1
    endif
    )
+ body.Statement.select(oclIsTypeOf(ComponentStatement))->iterate(cs : ComponentStatement; result: Integer = resultSimple |
    result + nestedNOS-Shallow(comp, cs.Body)
    )
```
Like the definition of the Local-NOS, the definition of the $NOS_{\text{Shallow}}^d (op)$ uses a helper operation. The helper operation $\text{nestedNOS-Shallow}$ is used to determine the number of statements in the body of the operation $op$ and recursively the bodies of the CompoundStatements in the operation (second part of the definition). Additionally, it follows up all method calls where only the statements of operations defined in the analyzed component are taken into account. The $\text{nestedNOS-Shallow}$ is then applied on the bodies of these operations (first part of the definition). Beside the number of statements of these bodies that are summed up recursively, the method call itself is counted as a single statement. Anyway, method calls to operations that are encapsulated by i-acquired or i-contained components are not further regarded in this category and are interpreted as simple statements. Therefore, method call statements invoking operations of i-contained or i-acquired components are defined to be 1.

In the definition of the $NOS_{\text{Shallow}}^d (op)$ we directly followed up each method call to operations that are contained in the component under consideration and apply the calculation of the number of statements on the respective bodies. As a consequence the number of statements of operations that are called may be calculated multiple times. A different way of obtaining the number of statements with respect to a particular operation is to reuse a local operation metric. In the following listing we define the Recursive Shallow Number of Statement metric $NOS_{\text{Shallow}}^r (op)$ on behalf of the local operation metric $\text{nos-Shallow-op}$ that is added to the class LocalOperationMetrics - keeping track of the values of all local operation metrics.

```python
def nos-cc(operation:Operation): Integer =
    nestedNOS-Shallow(self.Component, operation.Body)
```

```python
def nestedNOS-Shallow-Op(comp: Component, body:Body): Integer =
    body.Statement.select(oclIsTypeOf(SimpleStatement))->
    iterate (cs : SimpleStatement; resultSimple: Integer = 0 |
        if cs.oclIsTypeOf(MethodCall) then
            if comp = cs.Operation.Component
                then resultSimple +
                    cs.operation.LocalOperationMetrics.nos-Shallow-op + 1
    ```
In contrast to the \( \text{NOS}_{\text{Shallow}}^d(\text{op}) \) that directly follows up each method call to operations of the considered component, the \( \text{NOS}_{\text{Shallow}}^r(\text{op}) \) is defined on behalf of the \text{nos-shallow-op} that calculates the shallow number of statement of a particular operation. In contrast to the \( \text{NOS}_{\text{Shallow}}^r(\text{op}) \) that is defined in the context of the component, in detail it is defined for \text{ComponentMetrics}, the \text{nos-shallow-op} is defined in the context of the \text{LocalOperationMetrics}. Thus, it is defined for each operation and not only for the ones directly provided by the component.

Generally, metrics can be defined in a direct way, analogously to the definition of the shallow direct number of statements. However, it is necessary to determine whether metrics that are based on the invocation-chain can be defined in a recursive way. In case of the Shallow-NOS intuitively we would assume that it is possible to calculate the values for the operations of the invocation-chain in an isolated way and to sum up the respective values afterwards. Anyway, it is necessary to determine whether a shallow metric has the required theoretical properties to calculate the metric in a recursive way.
9. Hierarchy-aware Measurement

9.3.2. Theoretical Properties

As we change the perspective towards a shallow operation view it is necessary to slightly modify the relational system $U$. Although the relations remain unchanged, the underlying set $A$ needs to slightly be modified. Due to the shallow operation perspective, the elements of $A$ are defined to be the operations with respect to the encapsulating component ($op_{Comp}$). Anyway, the theoretical analysis regarding the first nine steps is equal to the analysis performed for the Local-NOS.

In the previous section we defined two different kinds of metrics regarding the Shallow-NOS. The $NOS^d_{Shallow}(op)$ directly calculates the number of statements by directly taking all the bodies into account that are affected when operation $op$ is invoked. The $NOS^r_{Shallow}(op)$, instead, is defined on behalf of a modified NOS - the nos-shallow-op - that adds up the number of statements referring to the nos-shallow-op of each invoked operation.

In terms of relational systems the $NOS^r_{Shallow}(op)$ performs a concatenation operation, where the $NOS^d_{Shallow}(op)$ is directly defined. In the $NOS^d_{Shallow}(op)$ the helper operation is applied on each operation of the invocation-chain individually, where the results are added up afterwards to obtain the Shallow-NOS. Anyway, the concatenation is a binary operation. To illustrate the difference between $NOS^r_{Shallow}(op)$ and $NOS^d_{Shallow}(op)$ in more detail we use a schematic representation (Figure 9.3) of the method calls that are performed for e.g. a component’s provided operation.

![Figure 9.3.: Followed up method calls](image)

In figure 9.3 we consider the operations that are called when the CompA’s operation opA is invoked. Operations are represented by half ellipsoid shapes that are publicly (solid shapes) or privately (dotted shapes) available.
In table 10.1 we give an example for the number of statements that these operations could have. We give an overview of the Local-NOS, the nos-shallow-op, and the number of method calls to operations that are defined in the component under consideration (MC).

<table>
<thead>
<tr>
<th>Operation</th>
<th>NOS</th>
<th>nos-cc-op</th>
<th>MC</th>
<th>NOS_{cc}(op)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CompA opA</td>
<td>9</td>
<td>53</td>
<td>1</td>
<td>53</td>
</tr>
<tr>
<td>CompA opB</td>
<td>13</td>
<td>44</td>
<td>2</td>
<td>-</td>
</tr>
<tr>
<td>CompA opC</td>
<td>5</td>
<td>26</td>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td>CompA opD</td>
<td>9</td>
<td>21</td>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td>CompA opE</td>
<td>7</td>
<td>12</td>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td>CompA opF</td>
<td>5</td>
<td>5</td>
<td>0</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 9.4.: Overview of the Statements affected by the invocation-chain of CompA’s opA

The $NOS_{shallow}(op)$ and the helper method $nos\text{-shallow-op}$ are defined on behalf of the Local-NOS. More specifically, the $nos\text{-shallow-op}$ enhances the Local-NOS by referring to the operations that are affected by invoking the “source operation”.

To assure that we can reuse the number of statements of an individual operation it is necessary to analyze whether the basic metric - the NOS - has an extensive structure.

As discussed in section 7.2.3 a relational system with a binary operation $\circ$, e.g. concatenation $\mathfrak{u} = (A, R_1, R_2, R_3, \circ)$, has an extensive structure if the theorem of Roberts and Luce holds:

$$a R b \iff f(a) > f(b) \text{ and } f(a \circ b) = f(a) + f(b)$$

The relations regarding $nos\text{-shallow-op}$ have the same properties as the relations defined for the Local-NOS. We focus on the relation has more statements than (R) and its corresponding relation in the numerical relational system $> (R')\mathfrak{u}$ is mapped to. R and R’ have been identified as having a strict simple order which implies that the first condition holds.

The second condition means that the result of mapping the concatenated operations $a$ and $b$ to the underlying set of the numerical relational system $\mathfrak{a}$, needs to be
equal to the result of summing up the mapping of the individual operations to $B$. Without loss of generality we use a graph based notion of the statements of an operation that are represented by nodes, where the transitions from one statement to another are represented by edges similar to the interpretation used to define the cyclomatic complexity. In contrast to the cyclomatic complexity where branch and loop statements are of fundamental interest (predicate nodes), the NOS treats these statements in the same way as any other. Thus, we neglect branch and loop statements in the following without loss of generality.

For our discussion on the second condition, we use the graph based notion of the two operations $op_E$ and $op_F$ (see figure 9.4). Referring to table 10.1 we define $op_E$ to consist of 7 statement, one of which is a method invocation of an operation defined by the component under consideration ($MC_c$)($op_F$). Further we define the operation $op_E$ to have statements and method calls to operations encapsulated by i-contained or i-acquired components. These, however, are not followed up by metrics of this category and are interpreted as single statements.

![Diagram](image)

Figure 9.4.: Statements in ClassA-opC and ClassB-opA following up the method calls

Basically we can consider method invocations in two different ways. First, we can interpret a method invocation as some kind of placeholder that is substituted by the software fragment being invoked. Second, we can interpret method calls as some
kind of pointer to an operation. In the first case replacing the method call by the
invoked software fragment implies, that the method call statement is not counted,
wherein contrast the interpretation of a method call as a pointer implies that the
method call needs to be considered as a statement. To distinguish between both
cases of concatenation operations we use $\circ_s$ for the first, and $\circ_p$ for the latter.

We start with the first case were the method call is substituted by the statement
of the operation being invoked. Except method calls to the component’s operations,
statements are summed up according to the definition of the NOS. In case of the
method calls, instead, the respective statement is substituted and thus it is not
taken into account for the sum. Depending on the $MC_c$ the concatenated $\circ_s$ fragment
has less or the same number of statements as would result by applying the metric
first on the individual operations and summing up the values afterwards:

\[
f(a \circ_s b) \leq f(a) + f(b)
\]

More precisely the $f(a \circ_s b)$ is $MC_c(a)$ - the operation-centric number of method invo-
cations of the operations defined by the component under consideration with respect
to $a$ - smaller than summing up the values of the individual operations without fol-
lowing up the method calls:

\[
f(a \circ_s b) + MC_c(a) = f(a) + f(b)
\]

But the left term $f(a \circ_s b) + MC_c(a)$ is exactly the second kind of concatenation $\circ_p$.
The method calls are considered as individual statement and are counted as such:

\[
f(a \circ_p b) = f(a \circ_s b) + MC_c(a) = f(a) + f(b)
\]

The concatenation based on the $\circ_p$ is assumed for the nestedNOS-Shallow-0p. Thus,
the nestedNOS-Shallow-0p is a realization of the NOS with respect to the method
calls on operations that are defined in the component under consideration. In terms
of measurement theory, the NOS is additive under the assumption of the $\circ_p$ con-
catenation, whereas it is not additive when the $\circ_s$ concatenation is applied. As
a consequence the $NOS'_{Shallow}(\circ_p)$ that uses the $\circ_s$ concatenation has an extensive
structure.
In this section we discussed in detail how to analyze whether a metric has an extensive structure using a graph based consideration. The discussion is equivalent for many other local metrics that are enhanced and considered in the shallow sense.

Basically, an extensive structure implies a ratio scale (see section 7.2.3.2). Anyway as counting is the only way to obtain the number of statements the nestedNOS–Shallow–Op and thus the $NOS^r_{Shallow}(op)$ and $NOS^d_{Shallow}(op)$ are absolute scales. Like the Local-NOS do the $NOS^r_{Shallow}(op)$ and the $NOS^d_{Shallow}(op)$ fulfill the criteria of a measure.

### 9.3.3. Indicator

The Shallow-NOS ($NOS_{Shallow}(op)$) is even more suitable indicator for determining the complexity of the functionality covered by a particular operation, as it completely follows up the method calls on operations in the component under consideration that are affected when op is invoked. In other words, the Shallow-NOS determines the complexity of the functionality that is covered by op and all of the operations encapsulated by the component under consideration that are affected when op is invoked, rather than determining the complexity of an individual operation.

Additionally taking the number of method calls into account, a combination of both determines the level of delegation in the component, the number of elements that are affected and so on. All of these issues give indications on the readability, the understandability and the learnability of the functionality’s realization.

### 9.3.4. Summary

In this section we discussed the category of shallow operation metrics. We focused on the metrics and underlying properties that result from an enhancement of the local metrics. We discussed the category of shallow metrics using the Shallow-NOS. The properties of the $NOS_{Shallow}(op)$ compared to the Local-NOS basically remain unchanged.

Anyway, we mainly turned our attention on the concatenation of operations resulting from method calls to operations of the component. We identified that depending
on how a method call statement is interpreted, two different kinds of concatenation operations exist. The first one, the $\circ_s$, interprets the method call in a way that it is substituted in the concatenation with the software fragment being invoked. The second case $\circ_p$, instead, interprets the method call statement as being a pointer.

In the analysis of the theoretical properties we realized that interpreting the concatenation as $\circ_s$ results in the Shallow-NOS not being additive, whereas the Shallow-NOS is additive in case of the $\circ_p$. For the deep operation metrics we use the latter as the concatenation operation.

The analysis of other measures and metrics of the category of shallow operation metrics that result from an enhancement of the local operation metrics is performed analogously.

9.4. Deep Operation Metrics

The local operation metrics focused on the content of an operation in a local way, which implies that method invocations are not followed up. The shallow operation metrics, instead, consider the “source operation” and all operations of the invocation-chain that are encapsulated in the component under consideration. Finally, the deep operation metrics consider the “source operation”, its invocations to operations encapsulated in the component under consideration and the operations invoked on other components of the composition hierarchy.

Definition (Deep Measurement). Deep metrics consider the composition hierarchy with the component under consideration being the root in an isolated way.

As discussed in chapter 5 Hierarchical Component Model a component can have two different kinds of required relationships to other components. We called the first one i-contained relationships to highlight that instances of these components are contained in the instance of the component under consideration. The i-contained relationships are used to define the instance composition hierarchies. I-acquired relationships, instead, are used to determine the relationships between the component under consideration and components outside the “core instance composition hierarchy”.

191
Based on the different association types in the composition hierarchy the deep operation metrics are further refined in deep-contained, deep-acquired and deep-complete operation metrics. The deep-contained operation metrics, for instance, only consider the method invocations to operations the component under consideration has an i-contained relationship to. The deep-acquired operation metrics, instead, only consider the method invocations pointing to operations that are encapsulated in i-acquired components. Finally, the deep-complete operation metrics do not make any difference between the association types of the composition hierarchy. Thus, they take all method invocations to operations encapsulated in components of the composition hierarchy into account.

For the discussion and definition of the deep operation metrics we use the component model defined in chapter 5 Hierarchical Component Model (see figure 9.5). The component model is enhanced by a single element - ComponentMetrics - that is used to collect the deep operation metrics.

It should be noted that the i-acquired and i-contained associations among components are defined on behalf of roles. This means that in the associations among components one defines the source and the other component plays the role of an i-contained or i-acquired component from the source’s perspective. Roles are used at this point to simplify the definition of OCL-based metrics.

### 9.4.1. Deep-Contained Operation Metrics

We start our presentation of deep operation metrics with the sub-category of deep-contained operation metrics. In contrast, to the shallow operation metrics, where only the method invocations that point to operations being encapsulated in the component under consideration are taken into account, the deep-contained operation metrics additionally follow up the method invocations to the operations being encapsulated by components the component under consideration has an i-contained relationship to.

**Definition** (Deep-Contained Metrics). Deep-contained operation metrics consider the composition hierarchy with the component under consideration being the root in an isolated way, where only those operations of the invocation-chain are taken into account that are encapsulate din the component under consideration and components it has an i-contained relationship to.
In figure 9.6 we schematically illustrate a composition hierarchy. We define three components that are i-contained in the instance of component CompA. CompB and CompC are directly i-contained in CompA, where CompD is i-contained in CompB and thus indirectly i-contained in CompA.
As in the previous sections we discuss the deep-contained operation metrics on behalf of an explicit metric - the Deep-Contained Number of Statements of an operation \((\text{NOS}_{\text{Deep-Contained}}(\text{op}))\). Based on the schematic composition hierarchy presented in figure 9.6 we give an example how a path through the affected operations could look. The “deep contained method invocations” that are affected when \(\text{opA}\) is invoked are illustrated in figure 9.7.

In figure 9.7 CompA is the component under consideration. It consists of two operations \(\text{opA}\) and \(\text{opB}\). If we consider \(\text{opA}\) as the operation we want to apply a particular deep operation metric on, first we need to identify the invocation-chain. Starting with \(\text{opA}\) it invokes \(\text{opB}\), which invokes \(\text{opA}\) that is encapsulated in CompB and \(\text{opA}\) encapsulated in CompC, and so on. If we continue to follow up the method calls we
9.4. Deep Operation Metrics

get the set of operations that are affected by the invocation-chain when opA of CompA is invoked.

Thus, we define the sequence of operation invocations after invoking opA as:

CompA-opA → CompA-opB → CompC-opA → CompB-opA → CompB-opB → CompD-opA → CompC-opA.

9.4.1.1. OCL-based Definition

In the following listing we define the Direct Deep-Contained Number of Statements metric of an operation \( \text{NOS}_{\text{Deep-Contained}}^d(op) \) in terms of the OCL:

```plaintext
context Body
def nestedNOS-DeepContained(i-acquired: Set(Components), body: Body): Integer
  = body.Statement.select(oclIsTypeOf(SimpleStatement)) ->
    iterate (cs : SimpleStatement;
      resultSimple: Integer = 0 |
      if cs.oclIsTypeOf(MethodCall)
        then resultSimple + 1
        else resultSimple +
          nestedNOS-DeepContained(i-acquired, cs.Operation.Body) +
          1
        endif
      else resultSimple +1
    endif
  ) +
  body.Statement.select(oclIsTypeOf(ComponentStatement)) ->
    iterate (cs : ComponentStatement;
      result: Integer = resultSimple |
      result + nestedNOS-DeepContained(comp, cs.Body)
    )

context Component
def i-acquiredComponents(): Set(Component) =
  self.i-acquired->asSet()

context ComponentMetrics
def nos-DeepContained(operation: Operation): Integer =
```

195
9. Hierarchy-aware Measurement

For the definition of the $\text{NOS}_d^{\text{Deep-Contained}}(op)$ we use two helper operations. The $\text{i-acquiredComponents()}$ operation is defined in the context of Components. It is used to determine the components that are i-acquired by the component under consideration.

Due to the rules we defined for composition hierarchies, components that are directly and indirectly i-contained in the component under consideration can only i-acquire those components that are i-acquired by the component under consideration or that are immediate peers (disciplined hierarchy rule). Further, “i-acquired components” could i-acquire components that are i-contained in the component under consideration under the assumption that the i-acquired component i-acquires the component under consideration. Such a relationship, however, would violate the inter-component invocation relationship and therefore is not allowed.

The set of required components ($S_{\text{Required}}$) is comprised of the directly and indirectly i-contained components ($S_{i-\text{contained}}$) and the i-acquired components ($S_{i-\text{acquired}}$). These subsets are disjunct: $S_{i-\text{contained}} \cap S_{i-\text{acquired}} = \emptyset$. This implies, that $S_{\text{Required}} = S_{i-\text{contained}} \oplus S_{i-\text{acquired}}$. Therefore, it is sufficient to identify the set of i-acquired components. Method calls to operations provided by components that are not contained in $S_{i-\text{acquired}}$ therefore need to be contained in $S_{i-\text{contained}}$.

In the second helper operation $\text{nestedNOS-Deep-Contained}$ two parameters are expected. The first one - i-acquired - expects the set of i-acquired components, where the second one expects the body the calculation should be performed on. Like the shallow definition of the helper operation - the $\text{nestedNOS-Shallow}$ - we iterate over all statements defined in the body. The second part of the $\text{nestedNOS-Deep-Contained}$ is equal to the second part of $\text{nestedNOS-Shallow}$. In the first part, instead, we iterate through all method call statements and identify whether the operations these method calls point to are encapsulated by components that are i-acquired by the component under consideration. If the respective component is contained in the set of i-acquired components we consider the method call as a single statement. Otherwise, the respective component is either i-contained in the component under consideration or it is the component under consideration itself. In this case we follow up the method call to the body of the operation being
invoked and apply the nested NOS-Deep-Contained operation on this body. Further, we consider the method call itself as a single statement as discussed for the shallow number of statements (applying the \( \circ_p \) concatenation operation).

### 9.4.1.2. Theoretical Properties

The shallow NOS - the \( NOS^d_{\text{Shallow}}(op) \) - only differs from the direct deep-contained NOS - the \( NOS^d_{\text{Deep-Contained}}(op) \) - in one thing. In contrast to the \( NOS^d_{\text{Shallow}}(op) \) that follows up the method calls to operations that are encapsulated by the component under consideration, the \( NOS^d_{\text{Deep-Contained}}(op) \) further follows up the method calls to operations that are encapsulated by its directly and indirectly i-contained components. This, however, has no influence on the theoretical properties. Therefore, the theoretical properties of the \( NOS^d_{\text{Shallow}}(op) \) and the \( NOS^d_{\text{Deep-Contained}}(op) \) are equivalent.

Unfortunately, it is not that easy to determine the recursive definition of the deep-contained NOS - the \( NOS^d_{\text{Deep-Contained}}(op) \). From the perspective of the component under consideration all the components that are contained in its instance composition hierarchy are directly and indirectly i-contained in it (see for example figure 9.7). Anyway, if each component of the composition hierarchy is considered individually, these can i-acquire and i-contain other components based on the rules that are defined in chapter 5 HIERARCHICAL COMPONENT MODEL.

Referring to the schematic example of a composition hierarchy illustrated in figure 9.6, CompB, CompC and CompD are i-contained in CompA. If we change the perspective towards CompB, CompB i-contains CompD, but it i-acquires CompC. As from CompA’s perspective CompC is i-contained, this implies that it is not directly possible to define the \( NOS^d_{\text{Deep-Contained}}(op) \) on the values calculated for its i-contained components. In other words, it is not possible to directly apply a concatenation operation on the deep-contained metrics. Anyway, in the next chapter we introduce the parametrized measurement that provides a suitable solution to this issue.

### 9.4.1.3. Indicator

In contrast to the Local-NOS, that determined the number of statements of an individual operation, the Shallow-NOS \( NOS_{\text{Shallow}}(op) \) indicates the complexity with
9. Hierarchy-aware Measurement

respect to the component under consideration. The $NOS_{Deep-Contained}^d(op)$ further follows up the method calls to operations that are directly and indirectly i-contained in the component under consideration. Therefore, the $NOS_{Deep-Contained}^d(op)$ is a suitable indicator for the complexity covered by the component under consideration and its i-contained components. If the component under consideration is the root of the composition hierarchy the $NOS_{Deep-Contained}^d(op)$ indicates the complexity of the functionality covered by the invocation chain of op with respect to the application.

As proposed for the shallow operation metrics, it is possible to use the $NOS_{Deep-Contained}^d(op)$ in conjunction with other metrics like the number of method calls to give an indication on the readability, the understandability and the learnability. In contrast to the shallow operation metrics that focused on a particular component do the respective deep-contained metrics have their focus on the internal aspects of the composition hierarchy. This means that they give an indication on the characteristics with respect to the internal aspects of an application/system.

Anyway, it is the ratio between the deep-contained complexity and the deep-complete complexity that indicates best the complexity covered by the “core” composition hierarchy related to the complexity of the virtual composition hierarchy with respect to a particular operation. We think that this delegation ratio has a major influence on the reliability of an operation. Delegation ratios are discussed in detail in the last section of this chapter.

9.4.1.4. Summary

In this section we gave an overview of the deep-contained operation metrics. In contrast to the shallow operation metrics that focused on individual components, the deep-contained operation metrics focus on the “core composition hierarchy”. We discussed the issues of the deep-contained operation metrics on behalf of the Deep-Contained NOS ($NOS_{Deep-Contained}(op)$).

The theoretical properties of the direct Shallow NOS - the $NOS_{Shallow}^d(op)$ - and the $NOS_{Deep-Contained}^d(op)$ are equivalent, as the metric is only enhanced to also take the method calls on operations provided by components that are i-contained in the component under consideration into account.
Unfortunately, it is not possible to apply the NOS recursively on the individual components that are i-contained in the component under consideration as the relations among these do not necessarily need to be i-contained as well. Due to the composition hierarchy and the rules we defined for the composition, i-contained components can not i-acquire components unless they are already i-acquired by their parents or are immediate peers. This means that components that are i-contained in the component under consideration can have i-acquired relationships to other components that are i-contained in the component under consideration. In the next chapter we give a suitable solution to this issue and propose a parametrized metric.

Basically, we discussed so far how to enhance local operation metrics to shallow metrics that are affected by the invocation-chain in the last section and how to enhance them towards a deep-contained metric. Although we discussed the enhancement explicitly for the number of statements other metrics are enhanced analogously.

9.4.2. Deep-Acquired Operation Metrics

The deep-contained operation metrics are based on the invocation-chain taking those operations into account that are encapsulated by the component under consideration and its directly or indirectly i-contained components. The deep-acquired operation metrics, instead, are based on the content of the operations that are encapsulated in the i-acquired components of the component under consideration, their underlying sub-hierarchies and their i-acquired relationships.

**Definition** (Deep-Acquired Metrics). Deep-acquired operation metrics consider the operations encapsulated in the components under consideration’s (directly and indirectly) i-acquired components that are invoked from the operations encapsulated by components residing in the core composition hierarchy that are affected by the complete invocation-chain.

As discussed in the previous section the set of components being affected by the (complete) invocation-chain can be categorized as being either i-contained or (directly or indirectly) i-acquired by the component under consideration. As also the i-contained operations can invoke the operations of the (directly or indirectly) i-acquired components, as illustrated in figure 9.7, it is necessary to identify the
9. **Hierarchy-aware Measurement**

complete invocation-path, before being able to apply deep-acquired operation metrics.

The theoretical properties of the deep-acquired and the deep-contained operation metrics are equivalent. Thus we do not go further into detail on the deep-acquired operation metrics at this point. The indication towards quality characteristics, however, are either equal to the indication of deep-contained or shallow metrics, e.g. being a suitable indicator for reliability, or are based on the delegation ratios which are discussed in detail in the last section of this chapter.

### 9.4.3. Deep-Complete Operation Metrics

Often operations require pieces of functionality that are provided by operations encapsulated in components that are either i-contained or i-acquired by the component under consideration. These operations are used to fulfill the “source operation’s” contractually specified obligations.

The deep-complete operation metrics do not distinguish between the association types among the components of the virtual composition hierarchy and the component under consideration and consider all components the component under consideration is associated to equally.

**Definition (Deep-Complete Metrics).** Deep-complete operation metrics consider all operations and their encapsulating components that are affected by the invocation-chain.

In figure 9.8 a schematic “virtual composition hierarchy” is illustrated. Based on the rules defined in chapter 5 HIERARCHICAL COMPONENT MODEL the root of the “core composition hierarchy” (instance of C) could have i-acquired relationships with the instances of component B and D. C further i-contains instances of F and G. Based on the disciplined hierarchy rule G could have i-acquired relations to the instances of B, D and F.

Like the other categories we discuss the deep-complete operation metrics with an explicit metric - the Deep-Complete Number of Statements \(\text{NOS}_{\text{Deep-Complete}}(op)\). The \(\text{NOS}_{\text{Deep-Complete}}(op)\) is an enhancement of the metrics we discussed in the previous sections. The \(\text{NOS}_{\text{Deep-Contained}}(op)\) only considered the number of statements...
9.4. Deep Operation Metrics

affected by the method calls to operations that are encapsulated by the component under discussion and its directly or indirectly i-contained components. The \( NOS_{\text{Deep-Acquired}}(op) \), in turn, only considered the number of statements affected by the method invocations to operations being encapsulated in components being i-acquired by the component under consideration. The \( NOS_{\text{complete}}(op) \), finally, follows up each method invocations, regardless where the operations the method calls point to are encapsulated in and considers all statements equally.

9.4.3.1. OCL-based Definition

The Direct Deep-Complete Number of Statements \( (NOS_{\text{Deep-Complete}}^d(op)) \) is defined in terms of the OCL as:

```oclmixed
context Body
def nestedNOS-Complete(body:Body):Integer = body.Statement.select(oclIsTypeOf(SimpleStatement))->$iterate (cs : SimpleStatement;
resultSimple: Integer = 0 |
if cs.oclIsTypeOf(MethodCall)
then
    resultSimple +
    nestedNOS-Complete(cs.Operation.Body) + 1
else resultSimple +1
endif
)
```

Figure 9.8.: Schematic Virtual Composition Hierarchy
9. Hierarchy-aware Measurement

The deep-complete metrics by definition treat all method invocations alike and follow them up independently if the operations these are pointing to are the component under consideration, its (directly and indirectly) i-contained or its (directly and indirectly) i-acquired components. In contrast to the deep-contained and deep-acquired metrics for which it is not possible to define a recursive version due to the different association types, a recursive version of a deep-complete metric like the Recursive Deep-Complete NOS $NOS_{Deep-Complete}^{Deep}(op)$ is defined as follows:

```python
context Body
def nestedNOS-DeepComplete-Op(body:Body): Integer
    = body.Statement.select(oclIsTypeOf(SimpleStatement)) ->
        iterate (cs : SimpleStatement;
            resultSimple: Integer = 0 |
            if cs.oclIsTypeOf(MethodCall)
                then resultSimple +
                    cs.operation.LocalOperationMetrics.nos-DeepComplete-op + 1
                else resultSimple + 1
            endif
        )
    +
    body.Statement.select(oclIsTypeOf(ComponentStatement)) ->
        iterate (cs : ComponentStatement;
            result: Integer = resultSimple |
            result + nestedNOS-DeepComplete-op(cs.Body)
        )

case LocalOperationMetrics
def nos-DeepComplete-Op: Integer

case ComponentMetrics
def nos-DeepComplete(operation:Operation): Integer =
```
The definition of the $NOS_{Deep-Complete}(op)$ is very similar to the definition of the Recursive-Shallow NOS. The $NOS_{Deep-Complete}(op)$ differs from the $NOS_{Shallow}(op)$ in the operations encapsulated in the component types (component under consideration, contained or acquired component) it follows up. Further, it refers to the nos-DeepComplete-Op that is defined in the LocalOperationMetrics that is used to keep track of the values being calculated for the respective operation.

### 9.4.3.2. Theoretical Properties

Enhancing a metric from the local operation, through the shallow operation to the deep operation category, we already identified whether a metric is additive in the shallow category. The invocation-chain based metrics being enhanced are not additive in the deep-contained and the deep-acquired category due to the different association types and perspectives of the affected operations of the invocation-chain. The deep-complete operation metrics do not care about the different instance-association types and each method invocation is followed up.

Basically, the shallow and the deep operation metrics differ in the method invocations that are followed up to the operations encapsulated in components the component under consideration is associated to and/or the component under consideration itself. The theoretical properties except additivity remain unchanged in the enhancement. In other words, if the deep-contained and the deep-acquired metrics are not additive as they are affected by the association types of the composition hierarchy, but the respective shallow metric is additive, then the deep-complete metrics are additive as well as they do not care about the different association types of the composition hierarchy.

### 9.4.3.3. Indicator

Like the other NOS-based metrics the $NOS_{complete}(op)$ is an indicator for complexity. The deep-complete consideration does not have the focus on the aspects covered by the composition hierarchy in a localized way but indicates the complexity of the complete realization of a particular functionality. This means that it is the complete
9. Hierarchy-aware Measurement

version of metrics like the NOS that give the best indication of a particular quality characteristic.

The complete metrics of a particular kind are of major interest, as they give an indication of an aspect on the complete invocation chain. For some quality characteristics such as readability it is of higher interest to give an indication based on the shallow perspective, where we focus on the readability of the core composition hierarchy. For others instead, like complexity, it is the complete invocation chain that gives the best indication of the complexity of an operation.

9.4.3.4. Summary

In this section we discussed the deep-complete operation metrics. After having discussed the shallow operation metric that have their focus on individual components and deep-contained operation metrics that have their focus on the “core composition hierarchy” the deep-complete operation metrics take the complete invocation chain into account. The theoretical properties of the shallow, the deep-contained, the deep-acquired and the deep-complete operation metrics that are based on invocation chains share the same theoretical properties, as they only differ in which method calls are followed up.

Although we use a particular invocation-chain based metric to discuss the deep-complete operation metrics, the discussion is applied analogously for any other metric of deep-complete category.

9.5. Delegation Ratios

In the presentation of the deep operation metrics we highlighted that the Deep-Contained NOS, for instance, gives a suitable indication of the complexity with respect to a particular operation - the “source operation” of the invocation-chain - covered by the component under consideration and its directly and indirectly i-contained components that are affected by the invocation-chain. The Deep-Acquired NOS, instead, is a suitable indicator for the complexity with respect to a particular operation that is covered by the operations of the invocation-chain encapsulated in components being i-acquired by the component under consideration.
Finally, the Deep-Complete NOS indicates the complexity with respect to a particular operation of the complete invocation-chain.

Generally it is difficult to interpret, for example, the indicated complexity based on the NOS. What does it mean that the Deep-Complete NOS is 20? Depending on the functionality a Deep-Complete NOS of 20 may be extremely bad, for example, if the functionality being measured is something like a getter operation. On the other hand, a Deep-Complete NOS could also indicate an extremely efficient realization. Absolute indications are mostly very context sensitive, which means that these need to be determined for each context individually. Relative indications, instead, are more independently of the underlying context. We discuss the operation-centric delegation ratios of complexity metrics more in detail in the following, as these give a very suitable indication of the reliability of a functionality in a relative way.

We define the Deep-Contained NOS related to the Deep-Complete NOS as the *Deep-Contained Delegation-Ratio of the NOS*:

$$
DelegationRatio^{NOS}_{Deep-Contained}(op) = \frac{NOS_{Deep-Contained}(op)}{NOS_{Deep-Complete}(op)}
$$

In contrast to an absolute NOS giving an indication for the complexity, the deep-contained delegation-ratio of the NOS gives a meaningful indication of reliability. A

$$
DelegationRatio^{NOS}_{Deep-Contained}(op) = \frac{NOS_{Deep-Contained}(op)}{NOS_{Deep-Complete}(op)}
$$

A *DelegationRatio* gives an indication of reliability based on the complexity covered by the component under consideration and its (directly and indirectly) i-contained components in relation to the complexity covered by the operations affected by the invocation-chain in total. A high ratio (> 65%) means that a major part of the complexity covered by the operations of the invocation-chain is covered by operations that are under control of the component under consideration. A low ratio (< 35%), instead, means that most of the complexity is covered by operations encapsulated by components the component under consideration is i-acquired to.

The *Deep-Acquired Delegation-Ratio of the NOS* is defined as:

$$
DelegationRatio^{NOS}_{Deep-Acquired}(op) = \frac{NOS_{Deep-Acquired}(op)}{NOS_{Deep-Complete}(op)} = 1 - DelegationRatio^{NOS}_{Deep-Contained}(op)
$$
9. Hierarchy-aware Measurement

Generally the component under consideration has limited control of which i-acquired components it is associated to. Typically components fulfilling the contract of the acquired functionality are associated to the component under consideration during runtime. Although components theoretically may fulfill the contract, it still remains risky to use them as it is unclear whether they fulfill the “expected functionality”. Thus, we think that unless the i-acquired relationships are not exhaustively tested, i-acquired components are less reliable compared to components being under control of the component under consideration.

In terms of the deep-contained delegation-ratio of complexity metrics like the NOS, it is more reliable to use a component’s provided operations that are mainly covered by the component under consideration and its i-contained components. A refinement of the deep-contained delegation-ratio is the shallow delegation-ratio. The shallow delegation-ratio determines the ratio of the complexity of an invocation-chain that is covered by the operations of the component under consideration being affected by the invocation-chain of the “source operation” op:

\[
\text{DelegationRatio}_{\text{NOS}}^{\text{Shallow}}(op) = \frac{\text{NOS}_{\text{Shallow}}(op)}{\text{NOS}_{\text{Deep-Complete}}(op)}
\]

Comparing the \(\text{DelegationRatio}_{\text{NOS}}^{\text{Deep-Contained}}(op)\) and the \(\text{DelegationRatio}_{\text{NOS}}^{\text{Shallow}}(op)\), the latter is always smaller or at least as high as the \(\text{DelegationRatio}_{\text{NOS}}^{\text{Deep-Contained}}(op)\). Anyway, the higher the \(\text{DelegationRatio}_{\text{NOS}}^{\text{Shallow}}(op)\) is, the more of the functionality is covered by operations of the component under consideration. This, in turn, means that more of the functionality is under direct control of the component under consideration and a misuse of operations is minimized.

Finally, we define the \(\text{DelegationRatio}_{\text{NOS}}^{\text{Local}}(op)\) which determines the ratio between the complexity covered by the operation and the complexity of the complete invocation-chain. Basically, the \(\text{DelegationRatio}_{\text{NOS}}^{\text{Local}}(op)\) can be determined for each operation affected by the invocation-chain individually and not only by the “source operation”. In this case we get a detailed statement how much each operation contributes to the complexity of the invocation-chain in a relative way. This, means that without referring to the realization we are able to determine critical operations. Critical operations could be those that provide a much higher functionality compared to the functionality of the other operations of the invocation-chain.
Anyway, referring to the different kinds of composition hierarchies, the type composition hierarchy from our view have a high influence on the estimation towards quality characteristics like reliability. A component being i-contained in the component under consideration mainly means that its underlying elements or the component itself are instantiated by the component under consideration. But this does not tell us whether the operations of the i-contained component have been designed for to cope with the covered functionality of the component under consideration. Basically, this is defined by the type composition hierarchy.

Delegation ratios are a promising starting point for the prediction of an invocation-chain's reliability. As additionally to the instance composition hierarchy the type composition hierarchy (not being in the focus of this thesis) has an overwhelming influence in the prediction of reliability we defer the discussion on delegation ratios to future research.

### 9.6. Conclusion

In this chapter we gave an overview of measuring aspects of hierarchical component based system from different perspectives. We started with the local operation metrics that focused on the issues concerned with an isolated consideration of an operation. The shallow operation metrics, instead, are an enhancement of the local operation metrics where additionally the invocation-chain to operations that are provided by the component under consideration is taken into account. The deep-contained operation metrics further take the operations encapsulated in components that are i-contained by the component under consideration into account, which are affected when an operation of the component under consideration is invoked. Finally, we discussed the properties of the deep-complete metrics, that take all of the operations and their encapsulating components into account that are affected by the invocation-chain of a particular operation.

We discussed each of the categories using an explicit metric. We used the number of statements to discuss how a category based metric can be defined, e.g. the \(NOS_{Shallow}(op)\). Anyway, other metrics of that kind is defined similarly. For each category we gave an OCL based definition of the category-based metrics. These are
9. **Hierarchy-aware Measurement**

easily modifiable towards other metrics of that kind. Further, we discussed the theoretical properties of each category using the category-based number of statements, where we applied the method that is proposed at the end of chapter 7 **SOFTWARE MEASUREMENT**.

Furthermore, we discussed the concatenation of operations for the Shallow NOS and the Deep-Complete NOS where we were able to identify a concatenation operation \( \circ_p \) such that the Shallow NOS and the Deep-Complete NOS have an extensive structure. This means that we were able to calculate the value of the metrics by first concatenating the operations and applying the metrics afterwards, or to apply the metrics on the individual operations and apply the concatenation of the metrics (adding up the values) afterwards \( (f(a \circ_p b) = f(a) + f(b)) \).

As discussed, it is not possible to identify an additive metric regarding the different kind of relationships (i-acquired and i-contained) to other components in the composition hierarchy. From the perspective of a component it may i-contain two component instances that in turn depend on each other. Considering the i-contained components individually, however, leads to the case that one is allowed to i-acquire the other one. In the next chapter we turn our attention on this issue and introduce parametrized measurement with which it is possible to define additive metrics for the deep-contained and the deep-acquired perspective.

Finally, we discussed how the different categories have an influence on the indication of quality characteristics. The category of local operation metrics considers an operation in an isolated way. An indication of quality characteristics that is purely based on an isolated operation is of limited expressiveness, as the functionality is mostly realized by the operation through delegating major parts of the covered functionality to other operations. Thus, we discussed how the category-based metrics have an influence on the quality characteristics, where for example the indication for reliability gets more expressive if the complete invocation-chain is taken into account.

Finally, we introduced special kinds of metrics - the delegation metrics - in the context of hierarchical component based systems, that take the different association types into account. These give a suitable indication of a quality characteristic based on the quality of a particular category compared to the indication addressing the complete invocation-chain. For example, the ratio between the Deep-
9.6. Conclusion

Contained NOS and the Deep-Complete NOS, gives a good indication of whether a major part of the complexity is covered by the component under consideration and its i-contained components or if the major part is delegated to its i-acquired components. Comparing two components providing the same functionality and having the same value for the Deep-Complete NOS we would basically prefer the one that provides more of the functionality by its own. Anyway, for the indication of reliability the type composition hierarchy has an overwhelming influence. The latter, however, is not further considered in this thesis, such that we defer the further analysis of delegation ratios to future research.
In the previous chapter we discussed the different categories of hierarchy-aware measurement. We started with the local operation metrics category, before we went into detail in the shallow metrics category. We focused on the shallow operation-centric metrics category that result from an enhancement of the local operation metrics, especially those that are affected by the operation invocation-chain. Finally, we discussed the deep metrics category and its sub-categories - the Deep-Contained, the Deep-Acquired and the Deep-Complete metrics - that address the different kinds of associations between the components of the composition hierarchy.

Basically there are two ways to apply a metric. In the first case we consider the invocation-chain of a particular kind directly - e.g. in the shallow consideration only the invocations to the component under consideration are taken into account. For example, two operations - \( opA \) and \( opB \) - are encapsulated in the component under consideration. Further \( opB \) is invoked by \( opA \). In other words, \( opB \) is concatenated to \( opA \). In terms of measurement theory the metric - function - is applied on the concatenated operations \((f(opA \circ opB))\). In the second case, the value of the concatenated operations can be obtained by calculating the metric for each element individually and add up the results afterwards \((f(opA) + f(opB))\).
10. Parametrized Measurement

To apply a metric on the operations individually and add up the values afterwards in order to obtain the value for the concatenated operations several properties need to be fulfilled. In measurement theory a function fulfilling these properties (see chapter 7) is defined to have an extensive structure.

Anyway, in the discussion of shallow measurement we have demonstrated that the concatenation operation has an influence on the additivity of a function. We considered method calls in two ways, one of which refers to the method call as some kind of pointer to the operation being invoked ($o_p$). The second one, instead, interprets method call statements as a place holder that is substituted with the content of the operation being invoked ($o_s$). We have shown that for our example - the $NOS_{CC}(op)$ - the second interpretation leads to an additive metric, where the first one does not.

10.1. Perspectives

Although we were able to define an additive Shallow NOS, we identified that it is not possible to define additive metrics in the conventional sense for the deep-contained and the deep-acquired metrics if the different kind of instance-association types in the composition hierarchy are taken into account. Unfortunately, the perspective of a component in the instance composition hierarchy has an influence on how the components that are contained in an invocation chain are interpreted. In figure 10.1 we give an example of a composition hierarchy and the different kinds of associations between the components defined from the perspective of each component individually.

The virtualRoot i-contains three component instances, which in turn i-contain instances of other components. In the shallow view all components of the sub-tree, with the root virtualRoot, are taken into account. From the perspective of the instance of C two component instances - instances of F and G - are i-contained in C. Further, C has two i-acquired relationships with instances of D and B. Unfortunately, D and B are i-contained in A, which means that it is not possible to apply a deep-contained metric on the operations encapsulated by C without taking the association types of C into account. For example:
Deep-Contained metrics that are related to the invocation-chain of an operation are not additive. Deep-Acquired metrics that take the different kinds of association types into account are not additive either, where Deep-Complete metrics do not care about the association types. Thus, these are additive if the respective shallow metric is.

As deep-contained and deep-acquired metrics are not additive, it is necessary to apply a metric on the complete invocation path directly, which in turn means that operations and encapsulating components may be considered multiple times. If we consider a component in an isolated way without referring to the composition hierarchy, an i-acquired relationship from its parent’s perspective could either be i-acquired if the parent i-acquires the component as well, or it can be i-contained if it is an immediate peer of the component under consideration.

A suitable solution for metrics not being additive due to the different interpretations of associations in the deep-contained and deep-acquired category, but that are
additive without taking this difference into account, e.g. the additive metrics of
the shallow category, is to define a parametrized version of the metric. To ensure
that we can define an additive version of the metric, we need to interpret the as-
sociations between the component under consideration and other components (that
are affected by the invocation of one of its operations) in terms of the cases that
result from the Disciplined Hierarchy Rule. To address both cases of this rule, the
parametrized consideration needs to consider the component’s i-acquired relationships as if they would be i-acquired or i-contained.

For the example introduced at the end of the last section we would need to consider
the i-acquired components that are associated to C and are affected by invoking opC
in four ways:

1. B and D are i-acquired by C ⇒ \( f^{B,D}_{Deep\text{-}Contained}(opC) \)

2. B is i-acquired, where D is i-contained by C ⇒ \( f^{B_{a},D_{c}}_{Deep\text{-}Contained}(opC) \)

3. B is i-contained, where D is i-acquired by C ⇒ \( f^{B_{c},D_{a}}_{Deep\text{-}Contained}(opC) \)

4. B and D are i-contained by C ⇒ \( f^{B_{c},D_{c}}_{Deep\text{-}Contained}(opC) \)

Taking all possible combinations on how the i-acquired components could be in-
terpreted into account, we are able to ensure the additivity of a metric in the deep
sense. A pre-condition for the parametrized consideration, however, is that the met-
ric has an extensive structure in the shallow category, but due to the different in-
terpretations of the component’s i-acquired associations (i-acquired or i-contained)
it has not in the deep-contained and deep-acquired category.

In the example the components B, C and D are directly contained in A. That A is
virtual in this example has no affect on the considerations. We assume that the
component A has an operation opA that invokes opC. Considering the associations
C has with other components that are affected by the invocation of opC in an iso-
lated way results in the four cases presented above. Since from A’s perspective the
components B and D that are affected by invoking opC are contained in A, we would
choose the value interpreting B and D as being i-contained (⇒ \( f^{B_{a},D_{a}}_{Deep\text{-}Contained}(opC) \)).
10.3. Drawbacks

In the example we used in the previous section to illustrate how the parametrized versions of a metric looks, we used a component C having two i-acquired relationships that are interpreted as they would be i-contained or i-acquired in the parametrized consideration. This distinction is made to address how the relations are interpreted from the perspective of a father component in the composition hierarchy. As we have shown in the example there are four combinations on how the two i-acquired components can be interpreted from the father’s perspective.

If three i-acquired components are affected in the realization of an operation, there exist eight possible combinations. If, in turn, four i-acquired components are affected it results in 16 combinations. More general, if \( n \) components are affected it results in \( 2^n \) combinations that need to be addressed.

With an increasing number of i-acquired components that are affected in an operation’s realization, the benefit from using an additive version of a metric compared to a direct version of it, decreases. However, the number of combinations that need to be taken into account at the various levels of the composition hierarchy basically may be optimized. However, how the parametrized consideration can be optimized is subject of the further research.

10.4. Parametrized Measurement in Practice

After having theoretically introduced parametrized measurement, we demonstrate how parametrized measurement is applied in practice in the following.

Our sample composition hierarchy consists of three composition levels and seven components in total. We define CompA to be the root of the composition hierarchy, that i-contains three components - CompB, CompC and CompD. Further, we define CompG to be i-acquired by CompA and by CompD, following the disciplined hierarchy rule. CompC has an i-acquired relationship to its immediate peer CompB. Further, it contains the two components CompE and CompF. CompE, in turn, has an i-acquired relationship to CompB and its immediate peer CompF. The resulting composition hierarchy is illustrated in figure 10.2.
Based on the composition hierarchy presented in figure 10.2 we define the invocation-chain of our sample operation opA that is encapsulated in CompA. The operation opA invokes opC of CompC. opC first calls opB of CompB. Afterwards, opC invokes opE of CompE, which first calls opB of CompB and afterwards opF of CompF. In the invocation-chain opF of CompF is invoked afterwards by opC. Finally, opA invokes opD of CompD, which in turn invokes opG of CompG.

Thus, the sequence of operation invocations after invoking opA is given by:

$$\text{opA} \rightarrow \text{opC} \rightarrow \text{opB} \rightarrow \text{opE} \rightarrow \text{opB} \rightarrow \text{opF} \rightarrow \text{opD} \rightarrow \text{opG}.$$  

The invocation-chain is illustrated in terms of the UML communication diagrams in figure 10.3. Without loss of generality the virtual root is not illustrated at this point.
After having defined the composition hierarchy and the invocation-chain of the sample invocation of opA, we are able to apply an operation-centric metric. As in the previous chapter, we use the category-based NOS for our considerations. Therefore, it is necessary to identify the number of statements of the affected operations. In table 10.1 an overview of the number of statements and the methods being called by the operation under consideration is given. The composition hierarchy, the invocation-chain and the number of statements of each operation are sufficient for the discussion of the Deep-Contained NOS, for instance.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Method Calls</th>
<th>Number of Statements</th>
</tr>
</thead>
<tbody>
<tr>
<td>CompA opA</td>
<td>CompC-opC, CompD-opD</td>
<td>27</td>
</tr>
<tr>
<td>CompB opB</td>
<td>-</td>
<td>16</td>
</tr>
<tr>
<td>CompC opC</td>
<td>CompB-opB, CompE-opE, CompF-opF</td>
<td>22</td>
</tr>
<tr>
<td>CompD opD</td>
<td>CompG-opG</td>
<td>31</td>
</tr>
<tr>
<td>CompE opE</td>
<td>CompB-opB, CompF-opF</td>
<td>43</td>
</tr>
<tr>
<td>CompF opF</td>
<td>-</td>
<td>26</td>
</tr>
<tr>
<td>CompG opG</td>
<td>-</td>
<td>17</td>
</tr>
</tbody>
</table>

Table 10.1.: Overview of the statements affected by the invocation-chain of CompA’s opA

The deep-contained operation metrics only considered the method invocations to the operations contained in the composition hierarchy with the component being the root. Method calls to operations to its i-acquired components are not followed up and are regarded as a single statement. As discussed in the last chapter it is not possible to define a recursive version of the operation-centric metrics in the deep-contained and the deep-acquired category. A component’s association types are defined from a local perspective. In the composition hierarchy of figure 10.2, for example, CompE has two i-acquired relationships (CompF and CompB). From CompC’s perspective, however, CompF is i-contained in it, where from CompA’s perspective even CompB is i-contained.

Therefore, we apply the recursive application of a deep-contained or deep-acquired metric on the parametrized considerations of each component being affected by the
10. Parametrized Measurement

invocation-chain. In the following we explain how the Deep-Contained NOS is calculated in terms of parametrized measurement.

First, we turn our attention to the components that do not have any i-acquired relationships. These are CompB, CompF and CompG. Obviously a parametrized version of these components does not exist, as they do not have any i-acquired associations. The number of statements of the operations of these components being affected by the invocation-chain is illustrated in table 10.2.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Number of Other Statements</th>
</tr>
</thead>
<tbody>
<tr>
<td>CompB opB</td>
<td>16</td>
</tr>
<tr>
<td>CompF opF</td>
<td>26</td>
</tr>
<tr>
<td>CompG opG</td>
<td>17</td>
</tr>
</tbody>
</table>

Table 10.2.: Non-parameterizable number of statements for the operations of CompB, CompF and CompG

Next we focus on the Components on the lowest level of the composition hierarchy that has i-acquired relationships. In our example, this is CompE that has an i-acquired relationship with its immediate peer CompF and with the “inherited” i-acquired component CompB. Applying parametrized measurement on CompE both of its i-acquired associations need to be considered as they would be either i-acquired or i-contained. The four possible combinations of the parametrized NOS are defined in table 10.3.

<table>
<thead>
<tr>
<th>Combination</th>
<th>i-acquired</th>
<th>i-contained</th>
<th>Number of Statements</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOS_{CompF,CompB_{Deep-Contained}} (opE)</td>
<td>CompB-opB, CompF-opF</td>
<td></td>
<td>45</td>
</tr>
<tr>
<td>NOS_{CompF,CompB_{Deep-Contained}} (opE)</td>
<td>CompF-opF</td>
<td>CompB-opB</td>
<td>43 + 16 = 59</td>
</tr>
<tr>
<td>NOS_{CompF,CompB_{Deep-Contained}} (opE)</td>
<td>CompB-opB</td>
<td>CompF-opF</td>
<td>43 + 26 = 69</td>
</tr>
<tr>
<td>NOS_{CompF,CompB_{Deep-Contained}} (opE)</td>
<td>CompB-opB, CompF-opF</td>
<td></td>
<td>43 + 16 + 26 = 85</td>
</tr>
</tbody>
</table>

Table 10.3.: Parameterized Deep-Contained NOS for opE of CompE
After having calculated the parametrized Deep-Contained NOS for the affected components of the third level of the composition hierarchy, we calculate the parametrized Deep-Contained NOS for the affected components of the second level of the composition hierarchy. The components being affected by the invocation-chain that are defined in the second level having i-acquired associations are CompC and CompD.

From CompC's perspective only CompB is i-acquired. This means that the parametrized version of the Deep-Contained NOS needs to be chosen where CompB is i-acquired or i-contained and all other affected components are i-contained (\( \Rightarrow NOS_{CompF,CompB_a,Deep-Contained}(opE) \) and \( NOS_{CompF,CompB_c,Deep-Contained}(opE) \)). The Deep-Contained NOS is then defined as follows:

\[
NOS_{CompB_a,Deep-Contained}(opC) = MC_{Statement} + NOS_{CompF,CompB_a,Deep-Contained}(opE) + NOS_{Deep-Contained}(opF) = 70 + 26 = 96
\]

\[
NOS_{CompB_c,Deep-Contained}(opC) = NOS_{Deep-Contained}(opB) + NOS_{CompF,CompB_c,Deep-Contained}(opE) + NOS_{Deep-Contained}(opF) = 16 + 85 + 26 = 127
\]

The parametrized Deep-Contained NOS for opC of CompC and opD of CompD are presented in table 10.4.

<table>
<thead>
<tr>
<th>Combination</th>
<th>i-acquired</th>
<th>i-contained</th>
<th>Number of Statements</th>
</tr>
</thead>
<tbody>
<tr>
<td>( NOS_{CompB_a,Deep-Contained}(opC) )</td>
<td>CompB-opB</td>
<td>96</td>
<td></td>
</tr>
<tr>
<td>( NOS_{CompB_c,Deep-Contained}(opC) )</td>
<td>CompB-opB</td>
<td>127</td>
<td></td>
</tr>
<tr>
<td>( NOS_{CompG_a,Deep-Contained}(opD) )</td>
<td>CompG-opG</td>
<td>31</td>
<td></td>
</tr>
<tr>
<td>( NOS_{CompG_c,Deep-Contained}(opD) )</td>
<td>CompG-opG</td>
<td>17 + 31 = 48</td>
<td></td>
</tr>
</tbody>
</table>

Table 10.4.: Parameterized Deep-Contained NOS for opC of CompC and opD of CompD
Finally, we turn our attention on the first level of the composition hierarchy, where except the root only the root’s i-acquired components are defined. In our example CompA is defined to be the root of the composition hierarchy, where CompA has an i-acquired relationship to CompG. All other components affected by the invocation-chain are i-contained in it. Thus, the parametrized Deep-Contained NOS of opA encapsulated in CompA is calculated as:

\[ NOS_{\text{Deep-Contained}}^{\text{CompG}}(\text{opA}) = NOS_{\text{Deep-Contained}}^{\text{CompB}}(\text{opC}) + NOS_{\text{Deep-Contained}}^{\text{CompG}}(\text{opD}) = 127 + 32 = 159 \]

\[ NOS_{\text{Deep-Contained}}^{\text{CompG}}(\text{opA}) = NOS_{\text{Deep-Contained}}^{\text{CompB}}(\text{opC}) + NOS_{\text{Deep-Contained}}^{\text{CompG}}(\text{opD}) = 127 + 48 = 175 \]

10.5. Conclusion

In the deep category and its sub-categories that were introduced in the last chapter, we realized that metrics having an extensive structure in the shallow category are not additive in the deep-contained and deep-acquired sense due to the different interpretations of associations in the composition hierarchy. In this chapter we introduced the parametrized consideration of the metrics referring to the composition hierarchy. After having discussed the properties of parametrized measurement, we gave a detailed example on how parametrized measurement is applied in practice.

The idea of a parametrized consideration in the deep-contained and deep-acquired category is a powerful solution to define additive metrics referring to the different kind of associations in the composition hierarchy. Unfortunately, a large number \( n \) of component associations that need to be interpreted for an individual component lead to \( 2^n \) combinations that need to be defined. Anyway, from our perspective the number of i-acquired associations, an individual component would need to deal with, are limited. Thus, we think that a parametrized consideration of metrics is a suitable solution for defining additive metrics referring to the composition hierarchy.
PART
IV
Epilogue
We close this thesis with this final chapter, where we first give an overview of the issues discussed in this thesis in section 11.1 SUMMARY. Afterwards we highlight our research contributions in section 11.2 RESEARCH CONTRIBUTION. Finally, we give an outline of future research activities in section 11.3 Future Research.

11.1. Summary

In this thesis we introduced hierarchy-aware measurement of software component composition hierarchies, where we focused on measurement of operation-centric metrics.

We started this thesis with an overview of common definitions of the term “software components” in the first chapter. We recognized that mainly all of the component definitions are based on interfaces. Generally interfaces can be interpreted as organizational element of the operations a component exposes (provided interfaces) and the operations organized in interfaces that are required to fulfill its contractually specified obligations. But it is more the component’s provided operations that allow other software entities to use a particular component. Due to the importance
of operations we decided to use a operation-centric definition of software components and to regard interfaces as purely organizational elements as they do not add further functionality to the component. Therefore interfaces are neglected in our considerations.

In the second chapter we highlighted the properties of common component models and their encapsulating frameworks and methodologies based on the categorization of Lau and Wang [LW05a]. We focused on the structural relationships between components in software composition and realized that there exist two categories: flat and hierarchical component models. Based on these categories and their subcategories we enhanced Lau and Wang's taxonomy of software component models.

For our research activities only the frameworks and methodologies with a hierarchical component model are of interest. Thus, in the third chapter we gave an overview of the most important frameworks and methodologies having a hierarchical component model. Basically all of the frameworks having a hierarchical component model focus on particular aspects of the development process of component based systems or a tailored to a particular domain. We informally identified that KobrA is able to cover the issues of these frameworks.

In the fourth chapter we introduced a common component model based on the impressions gained from and terms used in the frameworks and methodologies with an underlying hierarchical component model. However, we started the chapter by discussing the different kind of association types resulting in two different composition hierarchies: the instance composition hierarchy and the type composition hierarchy. Further, we defined three rules that are used to clearly define composition hierarchies.

As metrics basically are of limited expressiveness unless they are interpreted towards e.g. quality characteristics, we gave an overview of general and specialized quality models in the first chapter of the third part (chapter 6). Further, we gave an overview of evaluation processes that are concerned with the validation of quality models and the relationship between quality characteristics and metrics.

Unfortunately, no clear consensus of specific terms of software measurement is found in the literature. As we have shown, basic terms like software metric and
11.1. Summary

Software measures are interpreted in the same but also in different ways. In the 7th chapter we gave a clear definition of both and related terms.

Principles of measurement theory are applied in the field of software measurement. The definitions of Bollmann-Sdorra and Zuse [ZB89] are central in this field. But as Konrad [Kon91, Kon92] recognized they misinterpreted some terms and principles of measurement theory as proposed e.g. by Roberts [Rob79]. Therefore, we gave a clear definition of the basic terms. Further, we gave an overview of the basic terms of measurement theory that are necessary for our considerations, where we defined a method that is applicable for the identification of the properties of operation-centric hierarchy-aware software metrics.

Afterwards, we turned our attention on important software metrics presented in the literature. We started with an overview of traditional complexity metrics like the cyclomatic complexity of McCabe. After presenting the traditional complexity metrics we presented commonly used metric suites like the object oriented metric suite of Chidamber and Kemerer, the MOOD suite of Brito e Abreu et al. and the Halstead metric suite. Finally, we gave an overview of metric suites that address issues related to software components.

In the 9th chapter we introduced hierarchy-aware measurement in composition hierarchies of component based systems. We identified the three main categories of operation-centric composition hierarchy metrics- the local operation, the shallow operation and the deep operation metrics, where we discussed each category in detail. In each category we identified the basic properties, gave a definition of a sample metric (category-based Number of Statements) in terms of OCL and discussed how the category-based metric is interpreted towards the quality characteristics.

Sub-categories of the deep metrics, like the deep-contained metrics that are affected by the different association types (i-contained and i-acquired) between the components that are affected by the invocation-chain of an operation, are defined not to have an extensive structure. Therefore, we introduced parametrized measurement that solves this issue and introduces a method to define additive version of deep-contained and deep-acquired metrics by considering the i-acquired relationships of the locally considered component as it would be either i-contained or i-acquired.
11. Epilogue

11.2. Research Contribution

The research results presented in this thesis basically address issues related to measurement of component based systems. Our main contributions, however, are the following.

**Common Hierarchical Component Model:** One of our main contributions in this thesis is the definition of a common component model for hierarchical component based systems that is based on the impressions gained from the analysis of the most important hierarchical component models.

We identified two different kinds of association types that in terms of meta-modeling are defined in the instance view of the $M_0$ level and the template view of the $M_1$ level (instance-contained/acquired). Second, we identified the type-contained/acquired association types that is defined in the instance view of the $M_1$ level and the template view of the $M_2$ level.

Finally, we defined two types of composition hierarchies - the instance and the type composition hierarchy - that are based on the association types and three basic rules. The Virtual-Root, the Disciplined-Hierarchy and the Inter-Component Invocation rule are fundamental for the clear modeling of hierarchical component based systems.

**Measurement Method:** The second major contribution we achieved in this thesis is the introduction of a method for determining the theoretical properties of software metrics. The method is based on aspects of measurement theory as introduced by Roberts and Luce [RL68].

The method consist of fourteen steps in total. The first four steps have their focus on determining the relational system $\mathcal{U} = \{A, R_1, \ldots, R_n\}$ and its underlying order. The steps five to nine, instead, are used to determine an appropriate numerical relational $\mathcal{B} = \{B, R'_1, \ldots, R'_n\}$ corresponding to $\mathcal{U}$. The next three steps, steps ten to twelve, address the application of operations such as concatenation on $\mathcal{U}$ and $\mathcal{B}$. In the thirteenth step the underlying scale’s $(\mathcal{U}, \mathcal{B}, f)$ type needs to be determined, as the scale’s type has a major influence on the applicability of statistical methods. Finally, the fourteenth step is used to determine whether the mapping $f$ fulfills the properties of being a measure or if it only fulfills the properties of a metric.
11.2. Research Contribution

**Hierarchy-aware Measurement:** The core contribution of this thesis, however, is the introduction and analysis of hierarchy-aware measurement in composition hierarchies. We introduced a categorization and naming scheme for the operation-centric metrics that follow up the invocation-chain of an operation. We identified that basically, three categories - the local-operation, the shallow and the deep operation metrics - exits in this context. The deep operation metrics category, however, is further refined and three sub-categories - the deep-contained, the deep-acquired and the deep-complete operation metrics - were introduced.

The deep operation metrics category and it sub-categories are particular for composition hierarchies. The deep-contained metrics are affected by the invocation-chain of the “source operation”. The association types between the component under consideration and the components being affected by the invocation-chain have a major influence on quality characteristics like reliability. For instance, a component that covers most if its functionality by their own compared to a component that delegates most of its functionality to components it has an i-acquired relation to, from our view is less reliable. Therefore, we informally discussed the effect of the categories on the reliability of the composition hierarchy in terms of delegation ratio metrics.

**Parametrized Measurement:** The last major contribution is the introduction of parametrized measurement. Based on the categorization of operation-centric metrics for composition-hierarchies we identified that these do not have particular properties like an extensive structure due to the different association types in the composition hierarchies in the deep-contained and the deep-acquired category. To solve this issue we introduced parametrized measurement in composition hierarchies.

Parametrized measurement is a powerful solution to define additive metrics for composition hierarchies when the different association types between the components that are affected by the invocation-chain are taken into account.
11. Epilogue

11.3. Future Research

The research results presented in this thesis are fundamental for the analysis of hierarchical component based systems. The most emerging future research activities in the context of hierarchical component based system that are affected by our research results are:

**Metric Suite:** The most emerging research activity based on the results presented in this thesis is the definition of a metric suite addressing the particular issues of hierarchical component based systems. From our view should metrics of a specialized software metric suite in this particular context address the different association types between the components of the composition hierarchies. Further, should metrics of a specialized metric suite take the positions of components in the composition hierarchy and the importance of their underlying operations to fulfill a component's contractually specified obligations into account.

**Quality Model:** Basically metrics and metric suites can be thought of being of limited expressiveness unless they are interpreted towards more general aspects like quality characteristics. Quality models like the ISO/IEC 9126 are generally defined and need to be tailored to a particular domain and user group. Based on the research results of Khosravi and Guéhéneuc [KG05] need specialized quality models be tailored to a particular user group and be applied to and verified on a sample application.

Therefore, it is necessary to define and finally analyze specialized quality models for hierarchical component based systems. The definition of specialized quality models, however, includes the definition of a tailored metric suite or the adaptation of an already existing one to cope with the issues of hierarchical component based systems.

**Estimation Model:** Estimation models like the COCOMO are basically defined in a general way and are often not tailored to a particular context such as hierarchical component based systems. To improve these estimation models for a more suitable estimation in the context of hierarchical component based systems, specialized metrics need to be used. These, however, need to be tailored to the context, similar to the definition of a specialized quality model.
### Acronyms

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>WWW</td>
<td>World Wide Web</td>
</tr>
<tr>
<td>UML</td>
<td>Unified Modeling Language</td>
</tr>
<tr>
<td>OSM</td>
<td>Orthographic Software Modeling</td>
</tr>
<tr>
<td>COTS</td>
<td>Commercial off the Shelf</td>
</tr>
<tr>
<td>OMG</td>
<td>Object Management Group</td>
</tr>
<tr>
<td>NAF</td>
<td>NATO Architecture Framework</td>
</tr>
<tr>
<td>ADL</td>
<td>Architecture Description Language</td>
</tr>
<tr>
<td>PECOS</td>
<td>Pervasive Component Systems</td>
</tr>
<tr>
<td>CoCo</td>
<td>Component Composition Language</td>
</tr>
<tr>
<td>SOFA</td>
<td>Software Appliances</td>
</tr>
<tr>
<td>EBP</td>
<td>Extended Behavior Protocols</td>
</tr>
<tr>
<td>QoS</td>
<td>quality of service</td>
</tr>
<tr>
<td>KobrA</td>
<td>Komponenten-basierte Anwendungsentwicklung</td>
</tr>
<tr>
<td>COM</td>
<td>Component Object Model</td>
</tr>
<tr>
<td>OS</td>
<td>Operating Systems</td>
</tr>
<tr>
<td>EJB</td>
<td>Enterprise Java Beans</td>
</tr>
<tr>
<td>CCM</td>
<td>CORBA Components</td>
</tr>
<tr>
<td>OCL</td>
<td>Object Constraint Language</td>
</tr>
<tr>
<td>ECOOP</td>
<td>European Conference on Object-Oriented Programming</td>
</tr>
<tr>
<td>Acronyms</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>-------------</td>
</tr>
<tr>
<td>WCOP</td>
<td>Workshop on Component-Oriented Programming</td>
</tr>
<tr>
<td>CPD</td>
<td>Component Packing Density</td>
</tr>
<tr>
<td>CID</td>
<td>Component Interaction Density</td>
</tr>
<tr>
<td>SD</td>
<td>Software Design</td>
</tr>
<tr>
<td>LOC</td>
<td>Lines of code</td>
</tr>
<tr>
<td>MTTF</td>
<td>Mean time to failure</td>
</tr>
<tr>
<td>JCL</td>
<td>Job Control Language</td>
</tr>
<tr>
<td>CCM</td>
<td>Component Complexity Metric</td>
</tr>
<tr>
<td>CPC</td>
<td>Component Plain Complexity</td>
</tr>
<tr>
<td>CSC</td>
<td>Component Static Complexity</td>
</tr>
<tr>
<td>CDC</td>
<td>Component Dynamic Complexity</td>
</tr>
<tr>
<td>CCC</td>
<td>Component Cyclomatic Complexity</td>
</tr>
<tr>
<td>MOOD</td>
<td>Metrics for Object Oriented Design</td>
</tr>
<tr>
<td>MHF</td>
<td>Method Hiding Factor</td>
</tr>
<tr>
<td>AHF</td>
<td>Attribute Hiding Factor</td>
</tr>
<tr>
<td>MIF</td>
<td>Method Inheritance Factor</td>
</tr>
<tr>
<td>AIF</td>
<td>Attribute Inheritance Factor</td>
</tr>
<tr>
<td>PF</td>
<td>Polymorphism Factor</td>
</tr>
<tr>
<td>CF</td>
<td>Coupling Factor</td>
</tr>
<tr>
<td>SMO</td>
<td>Software Measurement Ontology</td>
</tr>
<tr>
<td>MTTF</td>
<td>Mean Time To Failure</td>
</tr>
</tbody>
</table>
Bibliography


Bibliography


234


Bibliography


Bibliography


Bibliography


