A holistic measurement model tailored to software development in a multi-language environment

Sara Bauer, MA, BSc

for the degree of Master of Science

Supervised by Prof. Dr. Michael Felderer

Leopold-Franzens-Universität Innsbruck
September, 2018
Abstract

Software metrics have become a crucial tool in measuring different aspects of the software development process. Their usage and expressiveness, however, are not without criticism. While the managerial side of software development wishes to scale a software’s success in one all-encompassing number, software itself is far too complex a construct as that it can be pinned down to one numerical value. This problem is exacerbated if different programming languages have to be compared: to avoid comparing the proverbial apples and oranges, the respective metrics have to be contextualized. This thesis presents a holistic measurement framework which is tailored to the software development processes in multi-language programming environments. To provide an appealing name, the model was called HOLLY - an abbreviation from HOListic Language Comparability model. The development of the HOLLY model occurred during a collaboration with an Austrian Health Care Systems Provider, and was practically assessed in a case study. Extending purely practical needs, however, the HOLLY model endeavored to present a holistic measurement model tailored to software development in multi-language environments. Three base requirements were important to achieve, namely (1) generality (2) flexibility and (3) separation of layers.

The assessment of the HOLLY model showed that the perspective of interpretation is crucial for validity of the outcomes. To receive an overview of the status of two programming languages or product lines, the derived indicators are an apt means; however, only the subsequent observation and interpretation of the underlying metrics enable the induction of suitable countermeasures. The need for interpretation (i.e. comparing two metrics directly) versus the need for abstraction (i.e. an abstract indicator value) here exhibit a continuous interdependency, which must be considered in the evaluation of the provided data.
# Contents

1 **Introduction** 1

2 **Background** 3
   2.1 Definition of Metrics and Classification Schemes 3  
      2.1.1 Objectives of Software Metrics 4  
      2.1.2 Classification Schemes 5  
      2.1.3 A Goal Based Framework 10  
      2.1.4 Software Quality and Measurement Information Models 12  
   2.2 Predictive Modeling 15

3 **Related Work** 17
   3.1 Research in Comparing Programming Languages Metrically 17  
      3.1.1 Code Metrics 18  
      3.1.2 Test Metrics 20  
      3.1.3 Process Metrics 21

4 **Methodology** 25
   4.1 Development of the Generic model 25  
   4.2 Approach 29

5 **Application of the HOLLY Model** 31
   5.1 Case Study: Layout and Process 31  
   5.2 Decision of Metrics and Composition of Derived Metrics 34  
      5.2.1 Operationalization Method 40  
   5.3 Gearing Factor Creation and Application 41  
      5.3.1 Complexity: Comparative Algorithm Analysis 42  
      5.3.2 LOC: Literature Review 45

6 **Tool Support** 47
   6.1 Outline 47  
   6.2 Tool Description 49  
      6.2.1 Provided Functions 49  
      6.2.2 Execution Functionalities 50
7 Results

7.1 Gearing Factor

7.1.1 Interpretative Options

7.1.2 Absolute Comparison: Gearing Factor Application

7.1.3 Relative Comparison: No Gearing Factor Application

7.2 Indicator and Comparability Factor

7.2.1 Indicator

7.2.2 Comparability Factor

7.3 Evaluation predictive modeling approaches

8 Discussion

8.1 General Remarks

8.2 Findings

8.3 Limitations

8.4 Outlook and further research

9 Conclusion

References
Figures

1. Software system dimensions. ........................................... 6
2. Example of deriving metrics from goals and questions. .......... 11
3. Quality model concepts. .................................................. 13
5. Results for size and complexity metrics at class level (C = Total no. of classes, P = Total no. of programs, Avg = Average no. of classes). ................................................................. 18
6. Comparison between coupling metrics at class level and system level. ................................................................. 19
7. Test coverage of popular programming languages. ................. 20
8. Bugs per KLOC. ............................................................... 21
9. Source Text Productivity in non-comment lines of code per total work hour. .................................................. 22
10. Summary of findings in cross-language comparability. ............ 23
11. Adapted ISO model. .......................................................... 26
12. Adapted ISO model closeup. .............................................. 27
13. Listing of metrics devised for case study. ............................ 35
14. Metrics template devised for case study. ............................ 39
15. Diagram of the tool work flow. ......................................... 48
16. Excerpt of the provided data set for calculated complexity metric product A. .................................................. 54
17. Retrieved metrics For Product A and B. ............................ 57
19. Metrics with gearing factor application. ............................ 59
20. Temporal complexity comparison. .................................... 60
21. Temporal coverage comparison. ....................................... 61
22. Indicator comparison. ..................................................... 63
23. Complexity comparability factor Product A. ..................... 65
24. Complexity comparability factor product B. ...................... 66
25. Coverage comparability factor product A. ......................... 67
Code listings

1 Call to R and normalization method. ............................... 50
2 Main Controller calls. .............................................. 51
1. Introduction

Measuring software quality can be seen as the world’s 21 century analog to the medieval study of alchemy: it is controversial, yet people strongly believe that it is achievable with the right methods. Software metrics have become a crucial tool in measuring different aspects of the software development process. Their usage and expressiveness, however, are not without criticism. While the managerial side of software development wishes to scale a software’s success in one all-encompassing number, software itself is far too complex a construct as that it can be pinned down to one numerical value.

The successful application of software metrics depends on asking the right questions, rather than expecting the right answers. In other words, gaining useful information from the results of software metrics requires an in-depth amount of knowledge of the application and its behavior. This problem is exacerbated if different programming languages have to be compared: to avoid comparing the proverbial apples and oranges, the respective metrics have to be contextualized.

The main objective of this thesis is to describe a holistic measurement framework which is tailored to the software development processes in multi-language programming environments. To provide an appealing name, the model was called HOLLY - an abbreviation of HOListic Language Comparability model.

What differentiates this model from previous work is the consideration of the entire software development process, in contrast to focusing on certain attributes (e.g. quality assessment, project management etc.). Furthermore, research has often considered multi-language environments impeding the expressibility of the developed model [49]. This thesis aims to overcome this obstacle by introducing appropriate gearing factors for the languages, derived both from literature and empirically. As a software development model is only truly holistic if it reflects past processes, monitors the present ongoings and also predicts future trends and developments, a predictive modeling component is also included.
To provide an overview, chapter two provides background information on the key technical concepts which are relevant for this thesis. Next, chapter 3 discusses related work which has been conducted in comparing programming languages metrically. Subsequently, chapter 5 outlines the methodology behind the HOLLY model, followed by its practical applications in chapter 5. Next, chapter 6 describes the tool which was constructed to facilitate data processing. The model’s results are then presented in chapter 7, followed by a discussion. Last, a conclusion is presented to round off the topic.
2. Background

The following chapter provides an overview of the core concepts relevant for this thesis, followed by a presentation of the related work already conducted in the selected areas. Moving from the general to the specific, section 2.1 serves as an introduction to the topic of software metrics and possible classification schemes in a single language environment. This is followed by an overview of research which derived software metrics in multi-language environments 3.1. Last, an introduction to predictive modeling techniques is provided. The three sections constitute the main pillars upon which the thesis is built both on a pragmatic and scientific view: sound knowledge on the versatility of software metrics aided in building a robust and scientifically founded model for software metrics. Second, establishing the status quo of work which has already been conducted for multi-language software metrics served as the starting point from which this thesis’s model was built, and also highlights its unique contributions in this area of research. Last, incorporating predictive modeling techniques has enlarged the scope of the model and aided in developing a holistic measurement model.

2.1 Definition of Metrics and Classification Schemes

First, a brief introduction to the notion of software engineering will ease the reader into the subject at hand, and contextualize further definitions. According to Fenton and Bieman, software engineering ”describes the collection of techniques that apply an engineering approach to the construction and support of software products” [25]. In contrast to computer science, which builds the theoretical foundations of creating software, the focus lies on producing a software product in a both controlled and scientific way. Both attributes are of great importance to the topic of this thesis: that is, the control of the software measurement process helps to standardize and reproduce results. A scientific approach here means that the methods are derived from scientifically valid procedures, and the results are interpreted in an accommodated manner.

Software engineering thus constitutes the larger frame in which the software measurement process is undertaken. In its simplest definition, a metric constitutes a standard of measurement through which certain qualities of a process or
product can be assessed. These qualities depend on the selected field of study, and can thus assume different definitions in different contexts. This also applies for software metrics, whose usage has gained considerable attention over the past decades [25]. It is also advisable at this point to clearly distinguish between measurement and calculation, as the terms relate to different procedures, both of which will be used in this thesis. Measurement, for one, is a direct quantification derived from the bare entity provided (e.g. the total amount of lines of code of a program); calculation, however, constitutes an indirect process in which measurements are combined into a qualified item to derive an interpretable attribute (e.g. average cost of each error in a program) [25].

2.1.1 Objectives of Software Metrics

Moving theory and technicalities aside, the objectives and purposes of software metrics shall be established in more detail. Software metrics constitute a vital asset in the software development process, as they provide an empirically sound (and quantifiable!) means of reasoning about a project’s attributes. Fenton and Bieman equate the usage of software metrics to a health care system, poignantly stating "how can you tell if your project is healthy if you have no measures of its health"? [25]. Health, in this context, can be interpretative as the desirably state of a project in the software development process should reside. By ratiocination, this enables software developers to control software projects, rather than to merely run them. This enables the whole process to become more dynamic as well, as one can recommend trends, describe the magnitude of necessary corrective action, and monitor a product’s attributes.

These attributes, however, can differ considerably between the parties involved in the software development process: from a managerial point of view, metrics such as cost, productivity, user satisfaction and product improvement are important. They provide insight into the profitability and economy of the project. To no surprise, a software developer will be more interested in testability, fault tolerance and the meeting of process/ product goals. Often, these two viewpoints are seen as irreconcilable; however, an attempt to unite both perspectives can be beneficial to both parties.\footnote{There exist considerable attempts to bridge this gap in the software development process. For more information, see [23].}
Much as the deployment of software metrics can be beneficial to a software project, their misuse can be equally fatal. There exist several possibilities through which a metric might be misused:

1. The metric is used in an over-simplified manner, without further analyzing explanatory potential (e.g. ignoring the mathematical function behind a code coverage measures, which has severe impact on the result) [39].

2. The metric is unveiled and interpreted in isolation, without contextualizing its emergence with other metrics and trends of the software development process [26].

3. Most literally, the metric is misused to explain the wrong causalities (e.g. equating lines of code with productivity). Especially the latter was conducted ambitiously before receiving reevaluation (for example, see [19]).

While research has since then surveilled software metrics more critically (see, for instance, [22]), scrutiny must not lead to a defamation of software metrics in general. Quite contrary, only a critical perspective releases the true potential of software metrics. On can even use the negative examples mentioned above to arrive proactively at features the software metrics need to fulfill: (1) practicability (2) expressibility and (3) applicability. As for practicability, any metrical assessment should be preceded by clearly specified objectives on behalf of the software engineer [25]. Unfortunately, practice has shown that this is not always the case, as metrics are retrieved based on their availability (i.e. specified through the tools in action) and also their simplicity (e.g. the infamous Lines of Code). Moreover, their expressibility depends on the correct interpretation of the bare numbers, which has to be conducted by an expert. Connected to the problem of expressibility, applicability determines the scope and context in which a measurement actually gains validity. The following sections aim to present classification schemes which cover these aspects respectively.

### 2.1.2 Classification Schemes

As already stated above, software metrics constitute a vital asset in the software development process. Their usage and interpretation depends strongly on
the selected perspective, splitting metrics functionally into three categories: \(^2\) informational, diagnostic and motivational \([41]\). For stakeholders, metrics provide valuable guidelines which can aid in future decision making processes \([45]\). Software developers, on the other hand, can utilize metrics to detect flaws in the source code which hinder an optimal progression of the software development process. In this context, metrics become an important tool for attaining quality management in the software development process \([24]\). Last, motivational metrics can influence and benefit all parties involved in software development, improving for instance the agile system development life cycle \([36]\). The above mentioned categories are by no means disjoint; that is, the functionality of a certain metric can contain more than one aspect, depending on the specific problem definition.

Coupling metrics more practically to the different dimensions of software, \([44]\) proposed a three dimensional model for software systems, the dimensions of which can be used to classify the corresponding metrics. Figure 1 shows the composition (taken from \([44]\), adapted and translated by the author):

![Figure 1: Software system dimensions.](image)

*Quantity* refers to measures which capture the size, scale or range of the software. Typical examples are Lines of Code (LOC) (see section 3.1.1) or the number of test cases necessary to cover a given use case. Drawn on the y axis, *complexity* measures describe the amount of relationships software entities have

---

\(^2\)Note that the categorization methods described below are by no means an exhaustive list of all software metric categorization methodologies. They were chosen to highlight delineation possibilities and provide a rough categorization scheme for the reader.
among each other. For instance, if a module has \( x \) connections to other elements or modules, this number draws conclusion on the module’s complexity [44]. Last, the \( z \) axis correlates to the quality of a software system. Quality is – as in so many other contexts – a highly interpretative and subjective term, and thus depends on a predefined norm a system has to fulfill. These norms can either stem from experience (i.e. undocumented code is tedious to comprehend) or are specifically formulated as a quota (i.e. a certain code convention practice).

The concepts presented above provide a certain room for interpretation; still, they present a rudimentary outline on how to estimate software metrically. Numerous classification schemes have emerged, deriving suitable categories from the dimensions 1. From a pragmatic standpoint, it is sensible to determine a target unit on which the measurements are performed. The most prominent unit here is of course the source code itself, exhibiting applicative potential for all of figure 1 dimensions. To guarantee optimal quality of the produced code, testing is necessary. Metrics such as the amount of tests cases and test case coverage can provide vital insights into the nature of the underlying code. Last, with the rise of inventive software development approaches such as Agile Development, the process itself has gained more attention. As these measures can estimate productivity, they have become an integral part in the software metrics’ repository (see, for instance, [41], [37]). To facilitate the structure of the literature review presented in the following chapter, the following categories were selected to fit specific targets of the software system.

1. **Code Metrics.** Metrics which deal with the analysis of the source code, i.e. lines of code.

2. **Process Metrics.** Metrics measuring the productivity and progress of software development, i.e. bug fix time.

3. **Test Metrics.** Metrics concerned with test progress and test coverage, i.e. test failure rate.

Each metric discussed in the literature can thus be attributed to one of the categories described above. To avoid complications later on, all metrics discussed are briefly explained below. The listing should help the reader understand the formalism behind each metric, as well as it should demonstrate the accuracy in which metrics must be defined in order to be comparable across different
programming languages. The definitions contain, unless otherwise specified, textbook definitions and are thus not cited further.

<table>
<thead>
<tr>
<th>Metric</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Branch Coverage</td>
<td>Assures that every branch alternative has been exercised at least once under some test</td>
</tr>
<tr>
<td></td>
<td>$\left( \frac{NumberOfDecisionsOutcomesTested}{TotalNumberOfDecisionOutcomes} \right) \times 100$</td>
</tr>
<tr>
<td>Bug (per lines of code)</td>
<td>Defined to be a problem detected during test or deployment, or a missing feature excludes syntax errors, type errors, and other errors caught by the compiler</td>
</tr>
<tr>
<td>Bugs per KLOC</td>
<td>Same as above, for 1000 lines of code</td>
</tr>
<tr>
<td>Coupling between Objects</td>
<td>Two classes are coupled together if one of them uses the other, i.e., one class calls a method or accesses an attribute of the other class. Coupling involving inheritance and methods polymorphically called are taken into account</td>
</tr>
<tr>
<td>Cyclomatic Complexity V(C)</td>
<td>Complexity of a section of source code is the number of linearly independent paths within it $E - N + (2 \times P)$</td>
</tr>
<tr>
<td>Term</td>
<td>Definition</td>
</tr>
<tr>
<td>-------------------------------------</td>
<td>-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Data Abstract Coupling (DAC)</td>
<td>Number of ADT’s defined in a class. ADT is a user defined data type which can be used without having the knowledge of how it is created. This coupling occurs when attribute of one class is object of another class (definition taken from [35]).</td>
</tr>
<tr>
<td>Defect (per lines of code)</td>
<td>Any error that causes an unplanned change in the code, including syntax errors, logical errors, and changes in variable names</td>
</tr>
<tr>
<td>Depth of Inheritance Tree (DIT)</td>
<td>The depth of a class within the inheritance hierarchy is the maximum length from the class node to the root of the tree, measured by the number of ancestor classes</td>
</tr>
<tr>
<td>Function Coverage</td>
<td></td>
</tr>
<tr>
<td>Line Coverage</td>
<td>Assures that all statements in a program are executed at least once under some test, calculated as: $\left(\frac{\text{NumberOfStatementsExercised}}{\text{TotalNumberOfStatements}}\right) \times 100$</td>
</tr>
<tr>
<td>Lines of Code (LOC)</td>
<td>The number of effective source lines of code</td>
</tr>
<tr>
<td>Message Passing Coupling (MPC)</td>
<td>Number of message sent to another class</td>
</tr>
<tr>
<td>Metric</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Number of Children (NOC)</td>
<td>Number of children counts the immediate subclasses subordinated to a class in the class hierarchy.</td>
</tr>
<tr>
<td>Response of a Class (RFC)</td>
<td>Size of the response set of a class. The response set of a class includes all methods that can be invoked in response to a message to an object of the class. It includes local methods as well as methods in other classes.</td>
</tr>
<tr>
<td>Time for realizing program</td>
<td>The time a programmer needs to realize a program</td>
</tr>
<tr>
<td>Time to Fix Defect</td>
<td>The time taken to fix a defect according to the phase where it was removed.</td>
</tr>
<tr>
<td>Weighted Measures per Class (WMC)</td>
<td>Sum of complexity of the methods which are defined in the class.</td>
</tr>
</tbody>
</table>

Table 2.1.1: Metrics discussed in literature.

2.1.3 A Goal Based Framework

The categories presented in the previous section represent the entities which can be subject to measurement; as already mentioned, however, the successful deployment of software metrics must be preceded by a sound selection. Several frameworks and models have emerged to reflect this process; however, most suited to the approach used in this thesis, a goal based framework shall be presented in more detail. First introduced by Basili and Perricone, and subsequently elaborated upon ([15], [13], [16] and [17], the so called Goal-Question-Metric Approach (GQM) selects metrics in accordance with the information...
you need to meet predefined *goals*. Organization is required to take steps in order to arrive at meaningful measures (taken from [14]):

1. It must specify the goals for itself and its projects.
2. It must trace those goals to the data that is intended to define the goals operationally.
3. It must provide a framework for interpreting the data and understanding the goals.

In a first step, this requires the identification of the overall goals of a project, product or even an entire organization. Subsequently, one has to pose questions whose answers must be known in advance in order to understand whether the goals are met. Last, each question is analyzed to determine the measures required to answer each question respectively. To put this into an example, Fenton and Bieman demonstrates the use of the GQM Approach by evaluating the effectiveness of using a coding standard (taken from [25]):

![Figure 2: Example of deriving metrics from goals and questions.](image)

After the goal (*evaluate the effectiveness of an organization’s coding standard*) is established, several questions can be posed, which reflect attributes of the goal. If one wants to know about the effectiveness of a coding standard, it is only sensible to differentiate between people who actually *use* the coding standard, and programmers that do not (*Who is using the standard?*). Next, the productivity of the coders is included as a question (*What is the productivity of the coders?*), to determine if the proposed coding standard exerts a positive (or potentially negative!) influence on the coding rate. As pure quantity will
not, by any means, be a hallmark of quality, this aspect must also be considered (What is the quality of the code?).

The underlying metrics to answer the predefined questions can then be multi-fold in nature, including proportionality, degrees of experience of the coders (e.g. familiarity of the programming language in years, etc.), or code size respectively. As demonstrated in figure 2, more measures are often needed to answer one question. In a similar manner, one measurement may be used for more than one question, in the figure above is the case for code size. As this goal tree grows from the root to its leafs, this ensures that only goal related questions are answered, without diverging or posing new questions in the middle of the structure.

The framework presented above provides much in the way of practicability and intuitivity. Fenton, however, warns about its caveats by underlying the importance of combining the measurements in a sensible way [25]. Taking again figure 2 as an example, if the goal is to measure coder productivity, this could be calculated in terms of effort per lines of code. This relationship, however, does not emerge from the tree itself, but from the consultation of models which relate metrics appropriately. The next sections presents a suitable model which fulfills this criteria and was utilized for this thesis.

2.1.4 Software Quality and Measurement Information Models

As mentioned above, a conceptual model is needed which can reflect the desired metrical relationships in a predefined context. To assess specific aspects of the software development life cycle, utilizing software quality measures is sensible. Such a software model thus comprises "the set of characteristics, and the relationships between them that provides the basis for specifying quality requirements and evaluation" [28]. Indeed, software quality models have received considerable attention, emerging in the early 1970's. Their long tradition and constant reinvention not only underlines their importance and practicability, but also reflect the constant chance software products undergo.

The emergence of the early models necessitated the creation of a unified terminology for the quality characteristics used in software product evaluation,

3A detailed historical timeline of Software Quality Models is presented can be found in[46].
which resulted in the publication of the ISO 9126 standard, now succeeded by ISO 25010 [31]. Here, software quality is decomposed recursively into characteristics and subcharacteristics [48]. This procedure is perpetuated until a base step is reached in which a characteristic can be measured in order to evaluate the software quality [47]. Figure 3 describes this approach (taken from [49]):

![Figure 3: Quality model concepts.](image)

Dedicated to the measurement of the quality of software products, ISO also produced an expanded version containing software quality models inventories of proposed measures for these models [10]. The application of a software quality model can be complemented by a framework which specifies the measurement planning, performance and evaluation process, materialized with the ISO information model (taken from [30]):
As figure 4 shows, each attribute undergoes a specific measurement method which produces a base measure unit. Two or more measures can then be combined using a measurement function, which results in derived measures. The latter can then be integrated into an analysis model to retrieve an indicator (i.e., a numeric value), the value of which can then be utilized to “explain the relationship between it and the information needed, in the language of the measurement user, to produce an Information Product for his [sic] Information Needs” [10].

As can be seen in figure 3 and 4 Software Quality Models and Measurement Information Models partly overlap; therefore, it can be advantageous to combine software quality models from the ISO 9216 standard with the practicality of the Measurement Information model from ISO 15939. Abran et al. proposed a
mapping of the two concepts into a hybrid model [10]. Its integration, however, would have exceeded the scope of this thesis; the model created here thus relies on the base concepts presented above, using hybrid solutions as creative stimuli (see chapter 4). The resulting model can thus be considered a Tailored Quality Model [46] as it influenced by the basic quality models, but considers a specific domain, in this case the software development process of multi-language programming environments.

2.2 Predictive Modeling

Using a simplified definition, predictive modeling is the process of statistically predicting future behavior [34]. Given its applicative potential, a myriad of fields profit from its use. In the context of computer science, predictive modeling is a prominent usage in data mining; however, specialized application scenarios are also present in the field of software engineering (see, for example [11] or [29]).

For the purposes of this thesis, regression analysis is most suitable, and will be discussed in more detail below 4. A type of statistical modeling, regression analysis constitutes a means to estimate the relationship among different variables.

At its core, regression analysis uses a regression model with three parameters and values:

- Unknown parameters, called $\beta$, constituting a scalar of vector.
- Independent variables called $X$
- Dependent variables called $Y$

relating $Y$ to a function of $X$ and $\beta$, written as

$$Y = f(X, \beta)$$  \hspace{1cm} (2.1)

**Linear Regression.** The most commonly used model, linear regression endeavors to model a relationship between two variables by fitting a linear equa-

\footnote{An extensive overview of possible predictive modelling techniques would transgress the scope of this paper. Gray and MacDonell provide a concise overview of predictive models devised for software metrics [29].}
tion to the data in question. The linear regression line is of the form
\[ y = \beta_0 + \beta_1 x + \epsilon \] (2.2)

with \( \epsilon \) being the unobserved random error [4].

Sometimes, the data under observation does not fit a purely linear equation, but indicates curvature. A common approach is to conduct a polynomial regression, in which the relationship between \( X \) and \( Y \) is modeled as an \( n \)th degree polynomial in \( y \). A general form for polynomial regression of the \( n \)th degree thus yields:
\[ y = \beta_0 + \beta_1 x + \beta_2 x^2 + \beta_3 x^3 + \ldots + \beta_n x^n + \epsilon \] (2.3)

A general pitfall of polynomial regression models can be the over fitting of data, which means that the model fits the data too closely. Therefore, special care must be taken to find the right balance between data fit and generality. One indicator for a model’s practicality is the residual, which constitutes the difference between a data given data point and the regression line (or curve) [6]. Their behavior can be a critical factor in choosing the right regression model, as demonstrated in section 7.3.
3. Related Work

3.1 Research in Comparing Programming Languages Metrically

The proper selection, deployment and analysis is exacerbated if more than one programming language and/or software development process is involved. Indeed, cross-language or cross-development comparability can be considered an additional dimension in the deployment of software metrics. Achieving metric comparability, however, depends on the projects under observation [38] and must therefore be tailored to the idiosyncrasies of the software product(s) in question.

Research comparing programming languages from a metrical point of view has been conducted for over 20 years, starting in the mid-nineties [20], [21]. All studies share the conception that comparability must be defined with respect to pre-selected metrics, which are automatically retrieved from available projects. Only then can the results be meaningfully compared and sensibly interpreted.

As for the programming languages involved, studies have concentrated on a variety of programming languages, sometimes reflecting the prevailing programming paradigm of the time [12]. With the rise of object oriented programming, languages such as Java and C++ also came under scrutiny, given their immense popularity (Java occupying the first, C++ third place in 2018’s TIOBE rating\(^1\)). The presented studies all focus on the comparison of Java and C++ metrics, with some of them also containing additional comparisons [43], [2] (which does, however, not compromise the expressiveness of the two main programming languages under observation).

As described in section 2.1.2, studies and outcomes were categorized according to the metrics the research focused on. An exhaustive list of all mentioned metrics can be found in section 2.1.2. The section is concluded with a tabular summary of the most important findings.

\(^{1}\)The TIOBE Programming Community index is an indicator of the popularity of programming languages. Updated once a month, the index is arranged by the number of skilled engineers world-wide, available courses and third party vendors [12].
3.1.1 Code Metrics

Code metrics comparing Java and C++ source code have been the subject of several studies. Software complexity has been studied by [35]. Applying a set of 23 metrics (among those, Weighted Methods per Class (WMC)) to 15 programs written in Java and C++, resulted in similar outcomes (average WMC for Java: 330; average WMC for C++: 3.04). The increase for Java is attributed to the fact that the complexity of the main function was discarded in C++ (as it is not part of a class). Figure 5 juxtaposes the results (taken from [35]).

![Complexity and Size Metrics (Class Level)](image)

Figure 5: Results for size and complexity metrics at class level (C = Total no. of classes, P = Total no. of programs, Avg = Average no. of classes).

Having a closer look at figure 5 provides interesting insights. The Number of Attributes (NOA), for instance is slightly greater for C++, attributed to the fact that it has fewer classes compared to Java [35]. A similar reasoning applies for the Number of Methods (NOM), where C++ also exhibits a slightly higher average value. Corresponding to the fact that Java is a pure object oriented language, it has a higher Response of a Class (RFC) value. Note that the divergences in the values in the examples above can always be retracted to syntactic differences which influence the final result. The inferences one can draw from the actual numbers are an important step in the utilization of comparative software metrics. In other words, the inductive reasoning necessary to explain numerical differences contains more explanatory power than the actual numerical difference. As will be shown in chapter 5, Java and C++ will be analyzed for their cyclomatic complexity explicitly using identical algorithm comparison.

Coupling metrics, such as Data Abstraction Coupling (DAC), and Message Passing Coupling (MPC) exhibit more salient differences across the languages,
as demonstrated in figure 6. Java’s increased values for CBO and MPC can be explained by the fact that the main function resides inside a Java class, thus boosting coupling; in C++, on the other hand, the main function is not part of a class, hence dependent coupling references are omitted. Again, the hiatus in the numerical data can easily be explained under closer observation. The appropriate usage and interpretation of these values, especially when combining different metrics to derive software quality, is key in deriving meaningful, expressive measures.

Figure 6: Comparison between coupling metrics at class level and system level.

Cross-Language Customized Metrics

To finish with one of the most rudimentary software metrics (and also one of the most disputed ones), Lines of Code (LOC) have been studied extensively, as early as 1995 [32]. Despite its simplicity and easy retrievable, its usefulness has always been controversial. To add feasibility, [32] studied LOC per function points. Function points are a software size metric that solely draws on a program’s functionality [43], and thus acquire equal expressivity regardless of the underlying programming language. Coupled to the aforementioned requirements, Java and C++ achieved the exact same LOC value, namely 53 [32]. The equality can here be ascribed to the fact that function point measurement already introduced a level of abstraction; the result then indicated direct

\[\text{For more information, see [18].}\]
comparability of the languages *given a modified metric* which discards language idiosyncrasies.

The idea of fine-tuning a software metric to enable cross-language comparability was also implemented by [27]. Instead of utilizing LOC metrics out of the box, the so-called *ABC metric* was introduced, guaranteeing stylistic independence and a mathematical foundation. The advantage of ABC is that it was specifically designed to reflect language peculiarities (such as initialization syntax), resulting in customized counting rules. Even though the results promised absolute comparability and thus accurate estimation of project development time and reliability, no other study has refined or elaborated on this proposal [27]. However, similar studies have emerged with individual counting rules for LOC customization (see [40]).

### 3.1.2 Test Metrics

The comparison of test metrics has received little attention, with only one study conducted by Google [2]. In a survey spanning over 650 different projects, test coverage (composed of branch, function and statement coverage)\(^3\) was determined for Java and C++ (besides Python, Go and Javascript). Figure 7 describes the results of the survey (taken from [2]):

<table>
<thead>
<tr>
<th>C++</th>
<th>Java</th>
<th>Go</th>
<th>JavaScript</th>
<th>Python</th>
</tr>
</thead>
<tbody>
<tr>
<td>56.6%</td>
<td>61.2%</td>
<td>63.0%</td>
<td>76.9%</td>
<td>84.2%</td>
</tr>
</tbody>
</table>

Figure 7: Test coverage of popular programming languages.

At first, the results might not offer much in terms of interpretational potential. The differences between the languages were mostly attributed to structural, paradigm and best practice differences. Moreover, it was also pointed out that the languages themselves varied in terms of coverage measurement precision and procedure [2]. The apparent difference between Java and C++ test coverage, diverging by 4.6% does not directly lead to the implication that C++ code has a worse degree of coverage than Java; the study above is a storybook example

\(^3\)No specification is given as to which amount of branch, function and statement coverage influenced the final results, only that "mostly statement coverage" was used [2].
of the fact that metrics alone, especially when applied to different programming languages, cannot provide explanations. In other words, the results must not be interpreted literally to demonstrate that different programming languages exhibit worse test coverage percentages; while it is more sensible to acknowledge the apparent difference in behavior, it is more fruitful to analyze the factors which influence the outcome.

3.1.3 Process Metrics

Process metrics, in this context, encompass metrics which cover the software process from an incremental and developmental perspective. Metrics also include human-bound behavioral components such as programmer productivity and defect rate occurrences. Comparing two real world development projects (one Java and one C++), [42] studied different forms of software productivity. Java yielded better results for almost all metrics under observation; for instance, the C++ program produces two to three times as many bugs per lines of code as a typical Java program 4 [42]. Additionally, C++ generated 15-30% more defects per lines of code. Figure 8 provides a listing of the results, here using KLOC (i.e. 1000 Lines of Code) as a reference.

<table>
<thead>
<tr>
<th></th>
<th>Defects</th>
<th></th>
<th>Bugs</th>
<th></th>
<th>No. of samples</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Mean</td>
<td>Std</td>
<td>Mean</td>
<td>Std</td>
<td></td>
</tr>
<tr>
<td>C++</td>
<td>82</td>
<td>25</td>
<td>18</td>
<td>8</td>
<td>7</td>
</tr>
<tr>
<td>Java</td>
<td>61</td>
<td>11</td>
<td>6</td>
<td>2.5</td>
<td>5</td>
</tr>
</tbody>
</table>

Figure 8: Bugs per KLOC.

The metrics presented above stand in direct relation to the time needed to fix a bug, (Time To Fix Bug). Again, Java only required half the time C++ needed to fix defects. Compile time defects in C++ took 6 minutes to fix on average, whereas Java defects could be corrected within 1.5 minutes. Similarly, bugs found in unit tests required 30 minutes of correction in C++, whereas only 15 minutes were necessary for bugs found in Java [42]. Additionally, productivity,

4Bugs per Lines of Code, as well as Defects per lines of Code can also be associated with Code Metrics. Due to the fact that the same paper also studied hourly defect rates and corresponding Times to Fix Defects, the metrics were included in this section.
measured in lines of code per minute was determined, with Java producing 0.28 to 0.95 more lines of code per minute. Interesting as the results may be, little to no conclusions were drawn from the findings, blaming programmer idiosyncrasies and different development environments for Java’s better score [42].

Productivity rates were also studied by [43], using source text productivity in non-comment lines of code per total work hour, as shown in figure 9.

Figure 9: Source Text Productivity in non-comment lines of code per total work hour.

The results presented in figure 9 neither prove nor disprove [42] findings; rather, they only show that Java features a larger productivity range than C++. Again, the different results can be attributed to different study environments and perspectives. [42] monitored one C++ programmer assigned to a project and one Java programmer over a period of 3 months. In contrast, [43] conducted a closed experiment on a small coding assignment in which the C++ and Java code was produced. The two studies are thus not readily comparable, as both the code generation phases and the number of participants differs.

Summary of Findings

In the following, the findings are summarized in figure 10. To facilitate understanding, a column was added for each metric to indicate if the desired metric should exhibit a low or high value. Trade-off indicating that a value can either be low or high, depending on the selection of priorities. For Depth of
Inheritance Tree (DIT), a low number implicates smaller complexity but, but also hinders code reuse through inheritance. By implication, a high DIT value ensures more potential for code reuse, but also entails a higher complexity. A similar reasoning can be applied for the Number of Children (NOC). Note that citations were omitted in the findings, as they are listed in the respective sections.

<table>
<thead>
<tr>
<th>CATEGORY</th>
<th>METRIC</th>
<th>GOAL</th>
<th>RESULTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>CODE</td>
<td>Depth of Inheritance Tree (DIT)</td>
<td>Trade-off</td>
<td>Higher in C++</td>
</tr>
<tr>
<td></td>
<td>Number of Children (NOC)</td>
<td>Trade-off</td>
<td>Higher in C++</td>
</tr>
<tr>
<td></td>
<td>Response for a Class (RFC)</td>
<td>LOW</td>
<td>Smaller in C++</td>
</tr>
<tr>
<td></td>
<td>Weighted Methods per Class (WMC)</td>
<td>LOW</td>
<td>Smaller in C++</td>
</tr>
<tr>
<td></td>
<td>Cyclomatic Complexity V (G)</td>
<td>LOW</td>
<td>Not significantly different</td>
</tr>
<tr>
<td></td>
<td>Coupling between Objects (CBO)</td>
<td>LOW</td>
<td>Higher in Java</td>
</tr>
<tr>
<td>TEST</td>
<td>Test Coverage</td>
<td>HIGH</td>
<td>C++: 56.6%, Java: 61.2%</td>
</tr>
<tr>
<td>PROCESS</td>
<td>Defects per Line of Code</td>
<td>LOW</td>
<td>C++ generates 15-30% more defects</td>
</tr>
<tr>
<td></td>
<td>Bugs per Line of Code</td>
<td>LOW</td>
<td>C++ 2-3 times more bugs</td>
</tr>
<tr>
<td></td>
<td>Time to Fix Defect</td>
<td>LOW</td>
<td>C++: 2 times longer</td>
</tr>
<tr>
<td></td>
<td>Productivity (Lines of Code Per Minute)</td>
<td>HIGH</td>
<td>Twice as productive in Java</td>
</tr>
<tr>
<td></td>
<td>Time for Realizing Program</td>
<td>LOW</td>
<td>Realization in Java takes 1/3 longer</td>
</tr>
</tbody>
</table>

Figure 10: Summary of findings in cross-language comparability.
4. Methodology

4.1 Development of the Generic model

The main objective of the model presented in this thesis is to describe a holistic measurement framework which is tailored to the software development processes in multi-language programming environments. What differentiates this model from previous work is the consideration of the *entire* software development process, in contrast to focusing on certain attributes (e.g. quality assessment, project management etc.). Furthermore, research has often considered multi-language environments impeding the expressibility of the developed model [49]. This thesis aims to overcome this obstacle by introducing appropriate gearing factors for the languages, derived both from literature and empirically (see section 5.3). As a software development model is only truly holistic when it reflects past processes, monitors the present on goings and also predicts future trends and developments, a predictive modeling component is also included.

To fulfill this aim, the models described in section 2.1.4 served as the basis from which a customized, or tailored model [46] could be derived. Figure 11 shows the generic model created for this thesis, mainly derived from the ISO model 15939. To provide an appealing name, the model is subsequently called the HOLLY - an abbreviation from HOListic Language Comparability model. The key aspects are summarized subsequently:
• **Focus on multi-language environments.** To highlight the innovative approach conducted for this thesis, figure 11 makes use of colors to facilitate the understanding. The two (or potentially, multiple) software products, here called *Product A* and *Product B* are colored in yellow and blue respectively. Here, the entity *Product* is equivalent with one programming language; for the sake of simplicity, we assume that only one programming language is used per product. This can, of course, be easily expanded for customization, by recursively enumerating separate sub products (i.e. programming languages) which make up a multi-component product.

• **Inclusion of the entire software development process.** For the sake of space, the sub steps which lead to the derived measures are omitted from figure 11. The attributes which eventually lead to the derived
measures can vary depending on the product, a factor which must be considered during the subsequent computation steps. Notice also, that the selected attributes are not exclusively properties of the programming language per se; as it was this thesis’ aim to reflect the entire development process, metrics regarding the software development process (such as *Time to Fix Bug*) can also be included. Divergences here can also give interesting insights into how different programming languages behave according to the pre-selected properties.

- **Derivation of an indicator and comparability factor.** After derived measures for different products (in yellow and blue respectively) are established, the results are fed into the Gearing Factor Application. As this is the most important step, figure 12 shows a closeup of the internal procedures taking place within the analysis model and subsequently the derivation of an indicator. All three steps will receive closer inspection:

![Figure 12: Adapted ISO model closeup.](image)

1. **Gearing Factor Application.** The different measures are again portrayed in yellow and blue respectively. The numbers represent the derived measures: the numbers 1 and 2 in yellow and the blue
signify that the derived measures represent the same concept (i.e. a *complexity metric*), whereas numbers without a matching partner indicate measures which were only retrieved for one product. This can be the case when different programming languages exhibit different properties which are not measurable in the other language (e.g. the *Coupling between Object Classes* in object-oriented languages, which apparently do not exist in non-object oriented languages). Nevertheless, these metrics can included, but most be compensated for (for instance, in the form of a constant).

Subsequently, the metrics must be adjusted according to a *Gearing Factor*, which serves as an adjustment factor between the products. Section 5.3 will elaborate on this point in more detail. The resulting metrics are thus harmonized, and highlighted in green (here, the products are still numbered but differentiated by an apostrophe, indicating two different harmonized products). Again, metrics which do not have an appropriate counterpart are not considered.

2. **Indicator.** As it was this thesis’ aim to consider the whole software development process, an indicator constitutes a suitable (yet, of course, simplified) means to compare different products during and after the software development process. Due to the emphasis on multi-language environments, metrics must be made comparable in a meaningful way. This exceeds the previous application of a gearing factor, as the metrics must be conflated in such a way that the result answers a certain information need, and depends on the predefined requirements (see section 5).

3. **Comparability Factor.** The indicator constitutes an abstract, static point of reference to compare the programming languages in question. However, it is also useful to create a comparability factor which can integrate data *dynamically* into the ongoing software development process. Such a factor serves as an estimation to enable ad-hoc analysis and enable constant monitoring in multi-lingual environments. As such, the factor a proactive purposes, as it enables immediate reaction to quality deviations.

Another important alteration of the original model was to put the Analysis
Model and the Indicator on the same level. This was done because the Indicator phase contains itself an analysis part which is necessary to derive the appropriate indicator function, described in chapter 7). Therefore, the application of Analysis Models becomes more a subsidiary process, for instance in the form of predictive modeling, which further aid the Interpretation of the results.

4.2 Approach

The development of the HOLLY Model occurred during a collaboration with an Austrian Health Care Systems Provider, and was practically assessed in a case study. Chapter 7) will provide an extensive overview of the procedure. Extending purely practical needs, however, the HOLLY model endeavored to present a holistic measurement model tailored to software development in multilanguage environments. Three base requirements were important to achieve, and are summarized in table 4.2.1:

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Generality</td>
<td>The presented model needed to exhibit a certain degree of generality to ensure more widespread deployability. By implication, this entails certain degree of abstraction, as the model should not specify the specific metrics or measures which should be retrieved. This also holds for the specification of the Gearing Factor as well as the Comparability Function, which must both be individually developed for the project in question.</td>
</tr>
<tr>
<td>Flexibility</td>
<td>The model must be flexible to potentially comprise several programming languages, which can have repercussions in all subsequent steps (e.g. more sophisticated Analysis Models, several Indicators etc.)</td>
</tr>
</tbody>
</table>
The layers of the model must be isolated in their purpose, function and execution. This specification directly results from the Goal Question Metric Approach presented in section 2.1.3. Each layer fulfills an isolated step within the development of a software measurement model; the layer itself already produces a meaningful information unit, which in subsequent steps is modified to arrive at a synthesized result. The layers as such can thus easily be modified, expanded or exchanged.

<table>
<thead>
<tr>
<th>Table 4.2.1: Requirements of the HOLLY Model.</th>
</tr>
</thead>
<tbody>
<tr>
<td>The following chapter will demonstrate the practical application of the HOLLY model which was conducted during a case study.</td>
</tr>
</tbody>
</table>
5. Application of the HOLLY Model

The following chapters document the detailed application of the HOLLY Model devised in chapter 4 in the form of a case study. To provide a contextual frame, the layout and the work process with the company will first be described. Next, the practical instantiations of the the HOLLY model will be presented in a bottom up manner:

- **Section 5.2 Decision of Metrics and Composition of Derived Metrics.** This includes the provision of a *Metrics Template* which was devised for the company. Moreover, the tools and mechanisms with which the measures were extracted and reprocessed are outlined.

- **Section 5.3 Gearing Factor Creation and Application.** This section demonstrates the creation and application of gearing factors which were devised for the product lines in order to ensure cross-language comparability.

- **Chapter 6 Tool Implementation.** This chapter presents an automation of the above two sections, and describes the tool which was implemented for the purpose of normalizing, aggregating and computing the derived metrics, together.

- **Section 7.2 Indicator Derivation.** Utilizing the reprocessed values described in chapter 6, this section describes the functions which were created to eventually derive a comparability factor for the product lines.

The presentation of the indicator derivation was purposely postponed until chapter 7, as the appropriate indicator function was derived empirically through the analysis of the reprocessed final data sets. To complete the model, the applied Analysis Models are presented in 7, followed by an interpretation. This concludes the process described in the HOLLY model.

5.1 Case Study: Layout and Process

The HOLLY model was practically evaluated through a case study in cooperation with an Austrian Health Care system provider. To retain anonymity, their two product lines will throughout this thesis be called *Product A* and *Product
B. Product A is written in Java, while Product B is written in C++. Additionally, the company endeavors to integrate a new, third product line, called Product C, which uses Swift [8]. The overall goal was to:

- **Goal.** Create an indicator system using software metrics for the software Products A & B to continually evaluate and optimize the developmental progress

- **Important Aspects.** Consider and appropriately integrate heterogeneous technologies and languages (Java, C++)

The research questions defined in chapter 4 cover these goals; however, their elaboration will extend the provided case study (see chapter 7).

The cooperation spanned a one-year period during which regular workshops were held. The workshops served to gain an understanding of the product lines, the developing environments and the questions which needed to be answered. In line with the Goal-Question Metric Approach presented in section 2.1.3, based on the goal, the following questions were posed for each product respectively, which helped in retrieving the appropriate metrics:

1. Which model can be used to retrieve the current technical state of the two product lines?
2. Which metrics are already retrieved during the development phase?
3. Which further metrics are sensible to retrieve during the development phase?
4. How can the metrics be synthesized and compared properly?

As it was believed that the capture of the entire development process will require iterative selection, deletion and reevaluation of useful attributes, the workshop incorporated feedback loops from all parties to reflect on the ongoing processes. This ensured that the company’s demands were met. A summary of the workshops is presented in table 5.1.1

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Type</th>
<th>Agenda</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Kick off meeting</td>
<td>Coordination of project goals and project work plan</td>
</tr>
</tbody>
</table>
## Introduction

Introduction into product landscape, tools, clarification of goal

- theory: present Concretion of Metrics
- practical: creation of Metrics Template

## Metrical Retrieval I

- finalize technicalities (frequency, additional tools..)
- finalize Metrics Template

## Metrics Retrieval II

- derivation of indicator system
- clarification and provision of missing data

## Data Preparation and Analysis Models

- presentation of refurbished data
- suggestions for tool development
- Presentation of Gearing Factor
- Presentation of refurbished data

## Finalization

- Presentation of tool
- sum up

---

### Table 5.1.1: Workshops held for use case study.

Next, table 5.1.2 outlines the tool environments used in the product lines respectively. Note that the specific tools will not be described in detail, but can be easily looked up; The figure is provided, however, for the sake of completion, and also to highlight the degree of heterogeneity which must be considered in multi-language programming environments:

<table>
<thead>
<tr>
<th>Cont. integration tools</th>
<th>Metric tools</th>
<th>Other</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Product A</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Gradle</td>
<td>Sonarqube</td>
<td>Zanata, Jira</td>
</tr>
<tr>
<td>Jfrog</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Artifactory</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Jenkins</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Apache Subversion</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| **Product B**           | SourceMonitor |       |
| Wheel                   |              |       |

- Jam
- ProjektWalker
- Visual Studio
- boost
- MySQL
Table 5.1.2: Tool environments for Product A & Product B.

Most important, for the purposes of this paper, were the Metric tools which were deployed so far to retrieve metric references. As described above, Product A uses the open-source platform Sonarqube surveil code quality, whereas Product B deploys SourceMonitor, a freeware program to monitor source code in numerous programming languages. After inspection, both tools were deemed apt for the purposes at hand, with some open-source suggestions to be considered for the future (see chapter 8).

5.2 Decision of Metrics and Composition of Derived Metrics

In the following, the metrics template devised throughout the workshops with the company will be presented. The metrics were chosen after careful examination of the development process, and resulted from intensive discourse with the company. The presented template is thus an amalgamation of empirical research (the inclusion of metrics which were positively noted in similar research) and individual selection (i.e. metrics which the company deemed desirable). As the proper selection and the subsequent retrieval of the metrics preoccupied more than half of the one-year working period, only a third of all selected metrics were retrieved and reprocessed (see chapter 8) The full template is graphically presented in figure 13. Due to these constraints, figure 14 presents only the fully-reprocessed extract of the finalized template.

\footnote{For simplicity, the metrics will from now on be referred to by the terms provided in figure 13.}
Figure 13: Listing of metrics devised for case study.
<table>
<thead>
<tr>
<th><strong>Cyclomatic Complexity</strong></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Description</strong></td>
<td>Measures the complexity of the source code.</td>
</tr>
<tr>
<td><strong>Object of measurement</strong></td>
<td>Classes of source code</td>
</tr>
<tr>
<td><strong>Scale</strong></td>
<td>Metric</td>
</tr>
<tr>
<td><strong>Method of measurement</strong></td>
<td>J aut. calc. with Sonarqube [&quot;complexity&quot;]</td>
</tr>
<tr>
<td></td>
<td>C aut. calc. with SourceMonitor [&quot;average_complexity&quot;]</td>
</tr>
<tr>
<td><strong>Frequency of Measurement</strong></td>
<td>Monthly</td>
</tr>
<tr>
<td><strong>Measurement Function</strong></td>
<td>Cyclomatic number on <strong>project</strong> level &amp; average over product</td>
</tr>
<tr>
<td></td>
<td>*average: over product line (all projects over product line)</td>
</tr>
</tbody>
</table>

**JAVA**

- Nr. of edges of the graph (E),
- nr. of nodes of the graph (N),
- nr. of connected components (P)

Def.: Measurement Function of control flow graph and subsequent calculation of complexity \( M \) with the formula:

\[
M = E - N + 2P
\]

**C++**

same
### Duplicate Blocks of Code *JAVA ONLY*

<table>
<thead>
<tr>
<th>Description</th>
<th>Determines the number of duplicated blocks of lines of source code.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object of measurement</td>
<td>Classes of source code</td>
</tr>
<tr>
<td>Scale</td>
<td>Metric</td>
</tr>
</tbody>
</table>
| Method of measurement | J aut. calc. with SonarQube ["duplicated_blocks"]  
C [potential 0-metric] |
| Frequency of Measurement | Monthly |
| Measurement Function | **Project** level & **average** per product  
Def. : “There should be at least 10 successive and duplicated statements whatever the number of tokens and lines.” |

### Comments

<table>
<thead>
<tr>
<th>Description</th>
<th>Determines the percentage of commented-out lines of source code.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object of measurement</td>
<td>Classes of source code</td>
</tr>
<tr>
<td>Scale</td>
<td>Metric</td>
</tr>
</tbody>
</table>
| Method of measurement | J aut. calc. with SonarQube ["commented_out_code_lines"]  
C aut. Calc. with SourceMonitor ["percent lines with comments"] |
| Frequency of Measurement | Monthly |
| Measurement Function | **Over project & average** over product line  
Def. from SonarQube:  
“Commented lines of code.” |

<table>
<thead>
<tr>
<th>JAVA</th>
<th>C++</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Documentation Comments

<table>
<thead>
<tr>
<th>Description</th>
<th>Determines the number of public APIs without comments header.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object of measurement</td>
<td>Classes of source code</td>
</tr>
<tr>
<td>Scale</td>
<td>Metric</td>
</tr>
<tr>
<td>Method of measurement</td>
<td>( J ) aut. calc. with Sonarqube [&quot;public_undocumented_api&quot;] ( \textit{doxygen} ) as a tool solution? ( C ) [potential 0-metric]</td>
</tr>
<tr>
<td>Frequency of Measurement</td>
<td>Monthly</td>
</tr>
<tr>
<td>Measurement Function</td>
<td>Def. from Sonarqube: &quot;Public API without comments header.&quot;</td>
</tr>
</tbody>
</table>

### Test Failures

<table>
<thead>
<tr>
<th>Description</th>
<th>Percentage of unit tests which fail when a particular test suite runs.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object of measurement</td>
<td>Unit tests</td>
</tr>
<tr>
<td>Scale</td>
<td>Metric</td>
</tr>
<tr>
<td>Method of measurement</td>
<td>( J ) aut. calc. with Sonarqube (1) [&quot;test_failures&quot;] + (2) [&quot;test_errors&quot;] ( C ) Calculation of errors if system does not built</td>
</tr>
<tr>
<td>Frequency of Measurement</td>
<td>Monthly</td>
</tr>
<tr>
<td>Measurement Function</td>
<td><strong>JAVA</strong></td>
</tr>
<tr>
<td></td>
<td>Def. (1): Number of unit tests that have failed with an unexpected exception.</td>
</tr>
<tr>
<td></td>
<td>Def (2): Number of unit tests that have failed.</td>
</tr>
<tr>
<td></td>
<td>( \text{skipped+failure}/(\text{success+failure+skipped}) )</td>
</tr>
<tr>
<td>Test Coverage</td>
<td></td>
</tr>
<tr>
<td>----------------</td>
<td></td>
</tr>
<tr>
<td><strong>Description</strong></td>
<td>Measures the amount of source code which is covered by unit tests.</td>
</tr>
<tr>
<td><strong>Object of measurement</strong></td>
<td>Classes of source code</td>
</tr>
<tr>
<td><strong>Scale</strong></td>
<td>Metric</td>
</tr>
</tbody>
</table>
| **Method of measurement** | J: aut. calc. with SonarQube [“coverage”]  
C: with gee: only libraries where a regression test exists (checks if all files are used; only relevant classes are measured, no generated classes), also called line coverage |
| **Frequency of Measurement** | Daily |
| **Measurement Function** |  
**JAVA**  
Coverage =  
\((CT + CF + LC)/(2*B + EL)\)  
CT = conditions that have been evaluated to 'true' at least once  
CF = conditions that have been evaluated to 'false' at least once  
LC = covered lines = lines_to_cover - uncovered_lines  
\(B = \text{total number of conditions}\)  
\(EL = \text{total number of executable lines (lines_to_cover)}\)  
**C++**  
Coverage =  
LC/ EC  
LC = covered lines (lines_to_cover - uncovered_lines)  
EL = total number of executable lines (lines_to_cover) |

Figure 14: Metrics template devised for case study.

Each metric carries a descriptive name, and also an indication if the metric is retrievable for both programming languages. **Description** offers a brief textual explanation of the metric’s purpose. **Object of measurement** ensures the specification of the measurement unit, as this could otherwise lead to incongruities. **Scale** signifies the unit in which the measure will be recorded. The exact retrieval method is denoted with the **Method of measurement** field, discriminating between the different programming languages. Next, **Frequency of Measurement** signifies with which frequency the measures will be retrieved. Most importantly, **Measurement Function** describes how single attributes of the software product chain (see 11) eventually lead to derived measures through specific measurement functions.
Retrieval and computation of happens automatically through the selected metrics retrieval tool: Sonarqube, for instance, uses the term “complexity” to denote Cyclomatic Complexity applying the formula described in figure 14. SourceMonitor, on the other hand, calls the same concept ”average_complexity”. To avoid computational discrepancies, the formulas behind the derived measures must be vigilantly checked and noted down. This also guarantees metrical equality in cases where the metrics can only be retrieved automatically for one programming language, and must be manually computed for the other (see, for example, Test Failures).

5.2.1 Operationalization Method

Given that the base step of the HOLLY model is the handling of entities suitable for further processing, this section discusses a systematic operationalization method to format the measures in a comparable way. The scientific validity of the presented method, mainly derived from [49], shall serve as the groundwork upon which all further, and more innovative, steps are built.

The operationalization of the measurements begins with the normalization of the retrieved values. This process ensures that the measurement data is comparable across different products [49]. Several normalization methods exist, which can greatly alter the form of the underlying numbers. By ratiocination, the selected normalization method can also negatively influence cross-language comparability, also acknowledged by [49]. While similar research has reverted to using more abstract measures, this thesis aims to counteract this problem by introducing gearing factors.

To assess the normalization method most suitable for this paper, the data underwent several iterations before the optimal normalization methods was determined. Optimal, here means that the data was reprocessed in a way which reflected and counteracted the idiosyncrasies of the data. Two salient issues were detected:

1. How should NaN (not a number) values be treated?
2. How should one deal with rows where each entry is identical?

For example, z-scaling [9] was one proposed one candidate method, converting
all indicators to a common scale with an average of zero ($\mu$) and standard deviation ($\sigma$) of one:

$$z = \frac{x - \mu}{\sigma}$$  \hfill (5.1)

Although otherwise promising, the normalization method would have necessitated the conversion to identical values to 0, which would have exacerbated the further handling of the data. Consultation with the company resulted in a positive feedback for Feature Scaling [33], which linearly transforms a value $x$ to a normalized value $y$ deriving the minimal value (min) and the maximum value (max) from the presented data set:

$$y = \frac{x - \text{min}}{\text{max} - \text{min}}$$  \hfill (5.2)

As a final solution for (1), it was decided that the values were not included in the calculation; for (2), identical row values were replaced with the mean of the eventual normalization method. Conclusively, identical values were replaced by 0.5.

### 5.3 Gearing Factor Creation and Application

The innovative contribution of this thesis lies in the derivation and creation of suitable gearing factors which enable more precise cross-language comparisons. As can be seen from figure 14, three different gearing factors had to be extracted, namely:

2. Gearing factor for Test Coverage.

For the sake of completion, a Lines of Code gearing factor between Java and C++ will also be discussed, which can be useful for the deployment of the expanded metrics template. Items 2 and 3 are measures which reflect on the programming language in an absolute numerical manner. In 2, the outcome reflects the percentage of code which is covered by (unit/R) tests. As for 3, the results indicate the percentage of tests which fail when a particular test suits
runs. Since the outcome depends for (2) on the quality and amount of test cases, and for (3) on the amount of test cases that fail within a particular test suit, one can derive a one-to-one correspondence between the two programming languages in question, resulting in a gearing factor of $I$.

For Cyclomatic Complexity (1), a specialized gearing factor had to be devised. Two observations lead to this conclusion: (1) results from previous research indicates that the complexity metrics for Java and C++ deviate from one another, even when applied for the same programming entity (see chapter 3). Reasons for that lie in different programming paradigms (e.g. in C++, the main function is not part of a class). (2) The analysis of the complexity formula might be the same, yet the language’s idiosyncrasies still influence and bias the outcome. The calculation applied for both programming languages to derive the complexity $M$ is:

$$M = E - N + 2P$$  \tag{5.3}

Here, $E$ denotes the number of edges in the graph, $N$ the number of nodes in the graph, and $P$ the number of connected components. These measures themselves are derived from a control flow graph which is created from the source code. As also backed by related research, the structure of the programming language – even if both belonging to the object-oriented paradigm – thus influences the construction and calculation of the resulting value $M$.

5.3.1 Complexity: Comparative Algorithm Analysis

As it is more important for this thesis to make programming languages comparable as a whole, the logical conclusion was to create a gearing factor by comparing programs dedicated to the exact same functionality, written in Java and C++ respectively. The programs’ complexity was then calculated with the same tools used in the use case study project, i.e. Sonarque and Source-Monitor. This ensures a direct correlation between the metrics derived from the two projects, and the gearing factor which was retrieved. Moreover, the utilization of relatively small programs written in the respective programming languages might also keep the programmer’s personal coding style at a relative minimum, as the desired function resolves around solving a concrete algorithm/
mathematical problem.

The thesis utilized an online github repository called *The Algorithms* [1] which provides a collection of algorithms implemented in several well-known programming languages, including Java and C++. Selecting from approximately 80 algorithms provided in both programming languages, a first filtering excluded algorithms which did not have a corresponding counterpart in the other programming language. The remaining algorithms were then run with different sample inputs, to ensure the correctness of their behavior in both normal and edge cases. Algorithms which did not meet these criteria were excluded.

Next, Sonarcube and SourceMonitor were installed on a test machine, with special attention to the fact that the tools were the same releases as used in the case study, as well running with the exact same configurations. The results were recorded in tabular form, as shown in table 5.3.1. Note that the names of the algorithms and their exact functionality can be retrieved from [1].

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>comp. Java</th>
<th>comp. C++</th>
<th>factor Java/C++</th>
</tr>
</thead>
<tbody>
<tr>
<td>Armstrong</td>
<td>5</td>
<td>4</td>
<td>1.25</td>
</tr>
<tr>
<td>BinarySearch</td>
<td>8</td>
<td>4.5</td>
<td>1.777777778</td>
</tr>
<tr>
<td>BubbleSort</td>
<td>8</td>
<td>6</td>
<td>1.333333333</td>
</tr>
<tr>
<td>CocktailShakerSort</td>
<td>11</td>
<td>6.33</td>
<td>1.737756714</td>
</tr>
<tr>
<td>Decimal To Binary</td>
<td>5</td>
<td>2</td>
<td>2.5</td>
</tr>
<tr>
<td>Decimal To Hexadecimal</td>
<td>3</td>
<td>4</td>
<td>0.75</td>
</tr>
<tr>
<td>Dijkstra</td>
<td>13</td>
<td>5</td>
<td>0.75</td>
</tr>
<tr>
<td>EggDropping</td>
<td>8</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>Fibonacci</td>
<td>7</td>
<td>4.5</td>
<td>1.555555556</td>
</tr>
<tr>
<td>GCD</td>
<td>5</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>InsertionSort</td>
<td>8</td>
<td>6</td>
<td>1.333333333</td>
</tr>
<tr>
<td>Knapsack</td>
<td>7</td>
<td>5</td>
<td>1.4</td>
</tr>
<tr>
<td>Kruskal</td>
<td>16</td>
<td>3</td>
<td>5.333333333</td>
</tr>
<tr>
<td>Algorithm</td>
<td>n</td>
<td>t</td>
<td>Complexity Factor</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>---</td>
<td>---</td>
<td>-------------------</td>
</tr>
<tr>
<td>LinearSearch</td>
<td>6</td>
<td>3.5</td>
<td>1.714285714</td>
</tr>
<tr>
<td>LinkedList</td>
<td>6</td>
<td>3</td>
<td>4.333333333</td>
</tr>
<tr>
<td>Longest Common Subsequence</td>
<td>18</td>
<td>5.33</td>
<td>3.377110694</td>
</tr>
<tr>
<td>Longest Increasing Subsequence</td>
<td>10</td>
<td>4.5</td>
<td>2.222222222</td>
</tr>
<tr>
<td>MergeSort</td>
<td>12</td>
<td>3.75</td>
<td>3.2</td>
</tr>
<tr>
<td>PrimMST</td>
<td>14</td>
<td>3.75</td>
<td>3.733333333</td>
</tr>
<tr>
<td>QuickSort</td>
<td>10</td>
<td>2.25</td>
<td>4.444444444</td>
</tr>
<tr>
<td>RadixSort</td>
<td>13</td>
<td>4.33</td>
<td>3.002309469</td>
</tr>
<tr>
<td>RodCutting</td>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>SieveOfEratosthenes</td>
<td>8</td>
<td>2.5</td>
<td>3.2</td>
</tr>
<tr>
<td>TernarySearch</td>
<td>10</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>TowerOfHanoi</td>
<td>3</td>
<td>2.5</td>
<td>1.2</td>
</tr>
</tbody>
</table>

Table 5.3.1: Complexity analysis in Java and C++.

The last column contains the factor which is retrieved by dividing the Java complexity value with the C++ complexity number in each row respectively. Last, each metric multiplicatively receives its appropriate gearing factor,

\[
\text{ComplexityFactor} = \frac{1}{n} \sum_{i=1}^{n} \frac{x_i}{y_i}
\]  \hspace{1cm} (5.4)

Where \(\frac{x_i}{y_i}\) denotes the division described above per row \(i\) (Java being \(x\) and C++ \(y_i\)), with a \(\text{ComplexityFactor} = 2.35992517\), which, to adhere to the chosen conversion direction (Java -> C++), must finally be \(\text{ComplexityFactor} = \frac{1}{2.36} = 0.4237\).

Now that appropriate factors have been retrieved for the derived metrics, the Code metrics as described in figure 14 will be adjusted with their appropriate gearing factors (abbreviated to \(gf\) in the following equations). This yields:
\[ Java_{\text{complexity}} = value_{\text{com}} \times (gf_{\text{complexity}}) \] (5.5)

\[ Java_{\text{complexity}} = value_{\text{com}} \times 0.4237 \] (5.6)

The derived gearing factor is applied unilaterally, as it was constructed from one source programming language and applied to one target programming language. It is important here to specify this flow previously and then adhere to this conversion direction (in our case, Java -> C++). An unilateral gearing factor multiplication can be beneficial as it reduces the amount of data that must be adjusted. However, if more programming languages need to be harmonized, this process might no longer be valid, which is also reflected in the HOLLY Model figure 12). Thus, the finalized counterpart metric will be

\[ C++_{\text{complexity}} = value_{\text{com}} \times (gf_{\text{complexity}}) \] (5.7)

\[ C++_{\text{complexity}} = value_{\text{com}} \] (5.8)

Note that the metrics test coverage and test failures are of the same form, only that their gearing factor is 1 in both cases, which is why it was not listed here explicitly.

### 5.3.2 LOC: Literature Review

A suitable Lines of Code (LOC) gearing factor between Java and C++ is briefly discussed in this section as it is useful for the deployment of the expanded metrics template proposed in chapter 5. Contrary to the approach described in section 5.3.1, an appropriate gearing factor was derived from previous studies found in literature, and was average and divided as described in the previous section.
<table>
<thead>
<tr>
<th>Author</th>
<th>Java</th>
<th>C++</th>
</tr>
</thead>
<tbody>
<tr>
<td>Laird and Brennan [37]</td>
<td>53</td>
<td>55</td>
</tr>
<tr>
<td>Jones [32]</td>
<td>50</td>
<td>53</td>
</tr>
</tbody>
</table>

| average               | 51.5 | 53  |
| gearing factor        | 1.0485 | $\frac{1}{1.0485}$ |

Table 5.3.2: Lines of Code numbers found in literature.
6. Tool Support

This chapter describes the tool which was constructed to facilitate the preparation of the metrics chosen in chapter 5. As the preparation methods have already been treated theoretically, the focus of this chapter lies in demonstrating the automated approaches which have been created to facilitate the reprocessing of the data. The refurbished data will then be presented in chapter 7, together with an analysis of salient features.

6.1 Outline

To evoke appropriate reprocessing methods, the used data format needed to be standardized. The data was exported in its default format from both Sonarcube and SourceMonitor; for simplicity, it was transmitted in tabular .csv file format. A complete data set for both programming lines received for a timespan of 8 month (April 2017 - November 2017); irregularities in the provision of data shortened the time period (see chapter 8). The data was extracted monthly and for all submodules within the respective product lines. To facilitate further processing, the files were sorted in ascending order. After this manual preparation, the data is now ready to be normalized using the tool. Figure 15 presents the work flow, which is explained in more detail in the following.
First, the initial data set is normalized according to a predefined normalization method. Note that some metrics must first be composed if they are not to be extracted in their derived form from the tool. Normalization is conducted for each sub module over the provided months. The data is then aggregated and averaged over all sub modules to yield a single value which will be the referential metric value used for further analyses. Note that the applied analysis models are not part of the automated tool, but must be manually fed into the analysis tools used; still, this step is listed to provide a holistic picture of the work flow.
6.2 Tool Description

The source code was compiled with Java 8 [3], and must contain one external library, namely Rserve[7], which is a TCP/IP server allowing Java to use facilities of R[5]. In all, the program comprises the four functionalities described in figure 15, which are now described from a technical point of view.

6.2.1 Provided Functions

For reasons of simplicity, Rserve must be started manually within RStudio or via the command line, so that the tool can make a local connection on the default port (A connection is opened automatically upon the initialization of the Controller class).

Because R provides excellent means of utilizing data structures (i.e. by using so called data frames, which offer a convenient way to iterate over data), this proved to be quite useful for data reprocessing. The following methods are provided:

- \texttt{min\_max\_normalization()}. The normalization formula used.
- \texttt{normalizeData(inputfile, outputfile)}. Normalizes the data by means of a normalization function.
- \texttt{averageData(inputfile, outputfile)}. Averages the data over months.
- \texttt{unit\_tests(allTests, failures, skipped, destinationFile)}. Calculates unit tests in form \([\text{skipped}+\text{failure}/(\text{success}+\text{failure}+\text{skipped})]\) (for Java only)
- \texttt{c\_coverage\_computation (covered\_lines, uncovered\_lines, executed\_lines, destinationFile)}. Calculates coverage with formula \(\frac{\text{covered} - \text{uncovered}}{\text{exec}}\).

In contrast to the Java unit\_tests the c++ metrics for R tests had to be computed in a more elaborate manner by parsing the given file, as no prior level of aggregation was provided. The document listed the states of the R tests ("skipped", "failed", "success") individually and for each month (i.e., there could be two entries for May, one for "skipped" with SUM 1 and one for "success" with SUM 30), so these idiosyncrasies had to be considered and stored ac-
cordingly. The actual implementation was conducted in `TestFailureParser.java` class.

### 6.2.2 Execution Functionalities

The tool has to handle a considerable number files and directories to store the default, normalized and aggregated data. To manage file coordination both efficiently and centrally, a `FileManager` class was implemented. The class is initialized with a `String startDirectory`, which contains the location of the folder holding all relevant files or file paths. By providing a source directory for the Java and C++ metrics, all further functions then iterate over the provided sets, thus reducing the the number of explicit function calls in case of a potentially growing metric collection.

The `RStatistics` class realizes the actual implementations making calls to R methods internally, which facilitates deployment:

```java
RConnection connection=null;

public RStatistics(String r_source) throws RserveException {
    this.connection = new RConnection();
    connection.eval(r_source);
}

public void normalize_data_no_params() {
    try {
        connection.eval("normalizeData()");
    } catch (RserveException e) {} 
}
```

Listing 1: Call to R and normalization method.

The implementation of a `Controller` class enabled the encapsulation of all necessary components within a single module. This allows the initialization and execution of the tool to work in a fully automated manner, provided that the data formats adhere to to a predefined format (see section 6.1).

49
A controller for each programming language is initialized and allocated to the required function calls. The range of functions as well as the programming languages can easily be extended or interchanged for further purposes, enabling modularity. Listing 2 shows the main function calls in the Controller class:

```
Controller java_controller = new Controller(
    javaResourceDirectory, rCode, javaResultDirectory,
    javaFinalDirectory);
Controller c_controller = new Controller(cResourceDirectory,
    rCode, cResultDirectory, cFinalDirectory);

TestFailureParser test = new TestFailureParser("test");

java_controller.NormalizeData();
c_controller.NormalizeData();

java_controller.GetAverage();
c_controller.GetAverage();
test.parseFile();
```

Listing 2: Main Controller calls.
7. Results

This chapter describes the results which were deduced by the practical application of the HOLLY model throughout the case study. First, the derived and aggregated metrics are listed in tabular form, accompanied by insights and comments. In accordance with the model, the respective gearing factors will be applied and analyzed in section 7.1, followed by the derivation of an indicator value.

In the following, the metrics selected in chapter 5 were reprocessed with the tool described in chapter 6. To present a visual aid, figure 16 shows an excerpt of normalized and averaged data of Product A. Note that the module names were replaced with a numerical listing to retain anonymity.
Figure 16: Excerpt of the provided data set for calculated complexity metric product A.
The results are comprised in tabular form for the two product lines A and B respectively (table 7.0.1 and table 7.0.2). For each metric, the normalization method (Norm.) is stated, here uniformly abbreviated as F. Scaling (Feature Scaling). Next, the aggregation and average mechanism is mentioned (Avg.) describing the reference unit over which the data was averaged. Same Value describes how identical values within a sub module were treated; last, the aggregated value is presented. Note that this value is derived by aggregating a chosen metric over all month provided for each submodule, and then averaging the resulting values to derive one single representative value for the metric representing Product A. By implication, the same procedure is conducted for Product B. As the final values all constituted small decimal values, a factor of 100 was multiplied to each value, and the result was rounded up. This facilitates the further handling of the metrical entities.

<table>
<thead>
<tr>
<th>Metric</th>
<th>Norm.</th>
<th>Avg.</th>
<th>Same Value</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Complexity</td>
<td>F. Scaling</td>
<td>submodules</td>
<td>0.5</td>
<td>0.3351 = 33.51</td>
</tr>
<tr>
<td>Test Coverage</td>
<td>F. Scaling</td>
<td>submodules</td>
<td>0.5</td>
<td>0.5196 = 51.96</td>
</tr>
<tr>
<td>Unit Test Failures</td>
<td>F. Scaling</td>
<td>submodules</td>
<td>0.5</td>
<td>0.0614 = 6.14</td>
</tr>
<tr>
<td>Duplicated Blocks of Code</td>
<td>F. Scaling</td>
<td>submodules</td>
<td>0.5</td>
<td>0.4939 = 49.39</td>
</tr>
<tr>
<td>Documentation Comments</td>
<td>F. Scaling</td>
<td>submodules</td>
<td>0.5</td>
<td>0.5446 = 54.46</td>
</tr>
<tr>
<td>Comments</td>
<td>F. Scaling</td>
<td>submodules</td>
<td>0.5</td>
<td>0.4932 = 49.32</td>
</tr>
</tbody>
</table>

Table 7.0.1: Product A results.

<table>
<thead>
<tr>
<th>Metric</th>
<th>Norm.</th>
<th>Avg.</th>
<th>Same Value</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Complexity</td>
<td>F. Scaling</td>
<td>submodules</td>
<td>0.5</td>
<td>0.4993 = 49.93</td>
</tr>
<tr>
<td>Test Coverage</td>
<td>F. Scaling</td>
<td>submodules</td>
<td>0.5</td>
<td>0.6888 = 68.88</td>
</tr>
</tbody>
</table>

53
Table 7.0.2: Product B Results.

As can be detected from the figures above, the domains of the values are comparatively similar for all metrics. A salient deviation could indicate either a severe computational error on behalf of the devised tools or a potential quality impediment in the product line. Based on the results above, both scenarios can be precluded; this releases the data for further inspection.

To gain an overview of the metrics retrieved so far, the two product lines were graphically reprocessed into a radar chart. Figures 17 and 18 display the multivariate data constructs, thus illustrating the current software developmental statuses. To highlight the influence of the gearing factor, Product A is displayed with and without gearing factor multiplication (denoted in yellow as Product A and in green as Product A with gearing factor respectively).

Figure 17 clearly captures the two product lines’ different metric statuses. While Product B spans a relatively equilateral triangle, Product A’s metrics are in sum smaller and better reflect the metrics’ ideal tendencies. That is, the values for complexity and test failures are smaller which is in line with their desired behavior to decline. Notice also that these two axes can be directly compared with regard to the different product lines/programming languages, as they were computed in an identical manner. Even though different computation methods were used for test coverage (see chapter 5), the metrics exhibit expressible results. This highlights an important aspect of the HOLLY model, as it demonstrates that the evaluation of multi-language software metrics can be fruitful even though identical reprocessing of the data is not achievable. What is more important than complete comparability is its contextualization in the bigger software development picture.

To demonstrate the model’s upward scalability and extendibility, figure 18 paints a 5-dimensional picture of the derived metrics, also including metrics which were only retrievable in one product line. The missing counterparts were added as constants to the respective other product line.
The addition of more software metrics paints a more diverse picture of the two product lines; however, it also demonstrates that the focus of analysis (i.e. a more diligent selection and comparison) can be crucial for the comprehensiveness of the model’s output. Selecting viewer metrics which exhibit more direct comparability points enables more acute evaluation of the entities under observation.

### 7.1 Gearing Factor

In the following, the gearing factors devised in section 5.3 will be applied to the metrics extracted above. As already described previously, the gearing factor will be applied unilaterally, in this case to the value of Product A.
Retrieved Metrics For Product A and B (with constants)

Given that the gearing factors for test coverage and test failures are defined through their respective computation method and are equivalent, those values remain the same. Figure 19 highlights the application of the gearing factor for complexity:

The complexity bars in figure 19 provide a crucial insight into the importance of gearing factors as devised in the HOLLY model. Contrasting Product A (yellow) with Product B (blue) before the gearing factor application would lead to the conclusion that Product A has a better complexity value at the
current development state, surpassing Product B by 33%. After applying the gearing factor to Product A, however, one can detect a more drastic divergence between the product lines, with Product A surpassing Product B now by 72% (rounded). The difference indicates a more drastic divergence between the two product lines which should receive close attention and induce countermeasures.

The direct comparison highlights that gearing factor application is essential when evaluation is conducted one-on-one, as an unfiltered juxtaposition can taint the interpretation of the presented metrics, and can thus induce the wrong level of counteraction. The following subsection will elaborate on this hypothesis and provide a more hands-on example.

7.1.1 Interpretative Options

7.1.2 Absolute Comparison: Gearing Factor Application

The importance of the appropriate gearing factor application becomes evident when analyzing the metric development from a temporal perspective. By dissecting the monthly development of the metrics, the direct juxtaposition between the raw and the gearing factor-revised values becomes evident. To put
this into an example, figure 20 shows a complexity comparison of the two product lines over a seven month period (April 2017 - November 2017):

![Complexity Comparison Product A and B](image)

Figure 20: Temporal complexity comparison.

Again, the different complexity values for Product A (in yellow and green) demonstrate that a raw interpretation of the data can lead to erroneous conclusions. Complexity values have changed drastically over the first three months of the investigated period, indicating a zealous effort to lower the overall program complexity. Product B, on the other hand, remained more constant, with the complexity slightly increasing in the second half of the observed period. These (separate) insights can already be useful when analyzing products in isolation; to ensure multi-language comparability, however, gearing factor application here is essential, which can be seen in Product A’s drastic shrink. The adaptation still reflects the steep decline in complexity, however, it ranks the product’s complexity entirely below the complexity values displayed by Product B. The raw (here meaning that the gearing factor was not applied) values here can definitely lead to wrong conclusions about the product’s development, which would severely impede its comparative validity and reaction capacity.
7.1.3 Relative Comparison: No Gearing Factor Application

This problem becomes exacerbated if a suitable gearing factor derivation was infeasible. As described in chapter 5, this was the case for the coverage metrics, were a computational derivation of a gearing factor (such as devised for complexity) was by nature not practical. The values have thus to be treated in their default form, contextualizing a perceptual salience accordingly. Figure 21 presents a monthly comparison of the coverage values here over a six month period.

![Coverage Comparison Product A and B](image)

Figure 21: Temporal coverage comparison.

What can still be derived from figure 21 – even with contextual caution– is product B’s consistency with regard to coverage. In contrast, Product A undergoes several in-and decreasing developments, culminating in a minimum coverage value in October. This demonstrates that, even though the numeric values might not exhibit absolute comparability, general trends and behaviors can still be captured in a relative comparison.

---

1The different time spans, here, do not reflect either a personal selection nor a inaccuracies on behalf of the author, but result from differences in the original data sets.
7.2 Indicator and Comparability Factor

7.2.1 Indicator

In order to arrive at an abstracted indicator value, a calculation method was devised which presents a cumulative, gearing-factor adjusted sum of the provided metrics. The resulting value constitutes an abstracted point of reference which can serve as a means to contextualize progress and deviations. The metrics which do not have a counterpart in the other programming language will be summarized as a counterpart constant (abbreviated cc) and added as a constant to its counterpart.

\[
I_A = \text{complexity} \times (gf) + \text{coverage} \times (gf) + \text{unit\_tests} \times (gf) + (cc) \quad (7.3)
\]
\[
I_A = \text{complexity} \times 0.4327 + \text{test\_coverage} \times 1 + \text{unit\_tests} \times 1 + (cc) \quad (7.4)
\]
\[
I_A = 33.51 \times 0.4327 + 51.96 \times 1 + 6.14 \times 1 + (153.17) = 225.7698 \quad (7.5)
\]

\[
I_B = \text{complexity} \times (gf) + \text{coverage} \times (gf) + \text{unit\_tests} \times (gf) + (cc) \quad (7.6)
\]
\[
I_B = \text{complexity} \times 1 + \text{coverage} \times 1 + \text{unit\_tests} \times 1 + (cc) \quad (7.7)
\]
\[
I_B = 49.93 + 68.88 + 47.79 + (153.17) = 319.77 \quad (7.8)
\]

To summarize the findings presented above, figure 22 contextualizes the relationship graphically.

Ideally, each indicator value should occupy half of the ring. By ratiocination, more product lines would split the ring in more entities (e.g. a third product line would implicate that each product should take up a third). Here, the duality is optimal to understand the presented indicator values and detect shortcomings which can be the subject of future adjustments. The imbalance in 22 can be reduced to the underlying metrics which were already discussed in section 7.1.

Again, it is important to note that the perspective of interpretation is crucial in the validity of the figure presented above. To receive an overview of the status
of the product lines in question, the derived indicators are an apt means; however, only subsequent observation and interpretation of the underlying metrics enables the induction of suitable countermeasures. The need for interpretation (i.e. comparing two metrics directly) versus the need for abstraction (i.e. an abstract indicator value) here exhibit a continuous interdependency, which must be considered in the evaluation of the provided data.

### 7.2.2 Comparability Factor

To easily integrate new chunks of data without applying the metrical preparation described in chapter 6, a factor was derived which enables fast comparability of new information units. This procedure serves as a means to adapt small data units into the bigger picture of the software development process at the given time. As stated before, the factor here serves as an estimation to enable ad-hoc analysis of the data and engage pro actively against harmful deviations. Two aspects have to be considered in its computation:

- **Rescale new entries.** New values must be contextualized within the data set which had already underwent homogenization. The preprocessed data set retrieved via the HOLLY model here serves as a point of reference.
Periodically (i.e. annually or semi-annually) the corpus (i.e. the point of reference) can thus be updated to include the previously dynamically integrated chunks of data.

- **Idiosyncrasies of the programming language.** This can be solved with the application of a gearing factor as described above, and can be multiplied accordingly.

To integrate new values into a data set dynamically, the indicator is taken as a point of reference, and the factor is computed as follows:

1. Retrieve an average value for each submodule within a metric.

2. Average these results to retrieve one aggregated final result for a metric (called Absolute Average $AA$).

3. Division of normalized values summed up in tables 7.0.1 and 7.0.2 (called Normalized Average $NA$) by $AA$

<table>
<thead>
<tr>
<th>COMPLEXITY</th>
<th>$AA$</th>
<th>$NA$</th>
<th>Comp. Factor</th>
<th>Gearing Factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Product A</td>
<td>328.3852</td>
<td>0.3351</td>
<td>0.0010</td>
<td>0.4237</td>
</tr>
<tr>
<td>Product B</td>
<td>2.5616</td>
<td>0.4993</td>
<td>0.1949</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TEST COVERAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Product A</td>
</tr>
<tr>
<td>Product B</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TEST FAILURES</th>
</tr>
</thead>
<tbody>
<tr>
<td>Product A</td>
</tr>
<tr>
<td>Product B</td>
</tr>
</tbody>
</table>
Table 7.2.1: Product B results.

\[ CF_A = complexity_A \times Java\text{ind}icator\text{factor}(0.001020379) \]  \hfill (7.9)
\[ CF_B = complexity_{C++} \times C++\text{ind}icator\text{factor}(0.19492657) \]  \hfill (7.10)

The factors ascertained above were applied to the provided data set to demonstrate its validity. Therefore, the raw, averaged monthly values were multiplied with the comparability factor derived in table 7.2.1. As test failures could not be broken down to monthly values (see chapter 5), its application is only demonstrated for complexity and coverage, while the third metric is discussed in chapter 8.

To start, figure 23 presents the complexity comparability factor for Product A:

Figure 23: Complexity comparability factor Product A.

Here, the normalized and averaged data points (original values Product A) can
be compared to the values obtained according to equations 7.9 and 7.10 respectively, the difference indicated as a green line. As for evaluation, it becomes apparent that the retrieved comparability value cannot reflect strong oscillations or changes in the provided data set. More harmonic results, however, are presented in its counterpart values from Product B, the original values and the reprocessed values here matching almost completely, presented in figure 24.

Two important conclusions can be drawn from these observations:

1. The derived comparability factor cannot – unless further equipped – reflect extreme changes in the data set.

2. Smooth, steady transitions however can be reflected (see figure 24) properly.

This reasoning also holds for the respective coverage comparability factors. Again, observation 2. holds for Product A, as the decrease in coverage is still reflected in the reprocessed data, as the change was not too severe or abrupt. The results are demonstrated in figure 25:

By contrast, figure 26 demonstrates again that a not otherwise adjusted gearing
factor cannot capture extreme oscillation in the provided data.

The devised comparability factors can constitute an apt "quick-fix" to integrate new data points dynamically during the software development process.
Its problem with oscillating values can arguably limit the correct insertion of new values; however, one can counter argue that rapid changes in metrics can always be accounted to limited periods in the software development process, and thus do not constitute the norm. For future research, this can function as an interesting incentive to cut out these non-characteristic phases from the comparability factor creation process. Moreover, these idiosyncratic phases can be considered in isolation, whereas for the majority of time a steady developmental process is assumed as default.

7.3 Evaluation predictive modeling approaches

The predictive modeling approaches presented in chapter 2 were applied to the given data sets, intended to find a model that can predict new values accurately. This can benefit the software development process immensely, and can guide further decision-making procedures.

Each data set underwent regression analysis (linear regression, quadratic regression, cubic regression) to determine which of the models is most parsimonious with the given data. For comparative, the residuals of the used models are listed in table 7.3.1. Salient differences are highlighted in color to demonstrate the impact of the chosen regression model.

To start, figures 27 and 28 present the complexity metrics for Products A and B respectively:
As can be seen from figure 27, it can be sensible to investigate more regression models, as one model might not fit every data set. Given that the complexity metrics decrease considerably over the months under observation, a linear regression model (red line in figure 27) does not capture this behavior correctly. By contrast, however, quadratic and cubic models (green and blue lines respectively) more closely fit the data’s behavior.
As for figure 28, linear regression is already an acceptable solution, while the quadratic regression might fit the data too well, a problem already discussed in chapter 2.

The same plots were devised for the coverage metrics of Product A and B, presented in figures 29 and figures 30.
Here, both products show that a linear regression does not constitute an optimal; similarly, however, the quadratic and cubic curves are identical.
The results presented above constitute valuable insights into the provided data sets, and demonstrate that a more intimate inspection can be beneficial for choosing an appropriate predictive model. By implication, this can have positive influences on the steps of the HOLLY model, as the appropriate analysis influences the better adaptation of the model to new data. To sum up, the residual errors are summarized in table 7.3.1.

<table>
<thead>
<tr>
<th>Complexity</th>
<th>Linear</th>
<th>Quadratic</th>
<th>Cubic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Product A</td>
<td>26.79</td>
<td>14.97</td>
<td>14.03</td>
</tr>
<tr>
<td>Product B</td>
<td>0.0027</td>
<td>0.0028</td>
<td>0.0019</td>
</tr>
</tbody>
</table>

Figure 30: C++ coverage regression plot.
<table>
<thead>
<tr>
<th>Coverage</th>
<th>Product A</th>
<th>0.8979</th>
<th>0.4373</th>
<th>0.4888</th>
</tr>
</thead>
<tbody>
<tr>
<td>Product B</td>
<td>0.0016</td>
<td>0.0012</td>
<td>0.0013</td>
<td></td>
</tr>
</tbody>
</table>

Table 7.3.1: Residual errors of prediction models used.
8. Discussion

Most research compares programming languages from a purely numerical point of view; often, however, it is more practical to understand how these differences can be actively incorporated into analysis processes to generate more adaptive comparability models. Therefore, the main contribution of the model presented in this thesis is to describe a holistic measurement framework which is tailored to the software development processes in multi-language programming environments. What differentiates this model from previous work is the consideration of the entire software development process, in contrast to focusing on certain attributes (e.g. quality assessment, project management etc.). Furthermore, research has often considered multi-language environments impeding the expressibility of the underlying model [49].

To fulfill this aim, a customized model was created which should reflect the idiosyncrasies of multi-language software development environments. Its key contributions are summarized in the following:

- Focus on multi-language environments.
- Inclusion of entire software development process.
- Derivation of an indicator and comparability factor.

The development of the HOLLY model occurred during a collaboration with an Austrian Health Care Systems Provider, and was practically assessed in a case study. Extending purely practical needs, however, the HOLLY Model endeavored to present a holistic measurement model tailored to software development in multi-language environments. Three base requirements were important to achieve, namely (1) generality (2) flexibility and (3) separation of layers. To provide a practical assessment of the models validity, the next section will present the most important outcomes of the case study.

8.1 General Remarks

Proper Selection of Perspective. The clear selection of perspective proved vital in the conception and application of the HOLLY model. This means that the level of abstraction necessarily increases with the selected level of the model
(using a bottom-up perspective). This also implies that one sacrifices numeric accuracy for aggregation (e.g. normalization and rounding of values). These two perspectives constitute extremes on a wide scale of interpretative options, all eligible in their own right.

On the one side of scale, a direct metric comparison necessitates the introduction of concrete gearing factors, as the results might otherwise be tainted. The direct comparison highlights that gearing factor application is essential when evaluating metrics one-on-one; an unfiltered juxtaposition can taint the interpretation of the presented metrics, and can thus induce the wrong level of counteraction. For the interpretation of aggregated metric values, however, complete comparability is not as important as the proper contextualization of the reprocessed metrics.

Again, it is important to note that the perspective of interpretation is crucial for the validity of the outcomes. To receive an overview of the status of two programming languages or product lines, the derived indicators are an apt means; however, only subsequent observation and interpretation of the underlying metrics enables the induction of suitable countermeasures. The need for interpretation (i.e. comparing two metrics directly) versus the need for abstraction (i.e. an abstract indicator value) here exhibit a continuous interdependency, which must be considered in the evaluation of the provided data.

8.2 Findings

In the following, the most salient findings are summarized. The findings mirror the steps of the HOLLY model, thus providing an evaluation of each stage. Moreover, the results are segmented into the metrics under observation.

Gearing Factor

Complexity
The validity of the gearing factor was shown when comparing the two product lines with and without application of an appropriate gearing factor (see figure 20). The adaptation still reflects the steep decline in complexity for Product A, however, it ranks the product’s complexity entirely below the complexity values displayed by Product B. The raw (here meaning that the gearing factor was not applied) values here can definitely lead to wrong conclusions about the
product’s development, which would severely impede its comparative validity and reaction capacity.

**Coverage**
While the values of Product B remained steady over the observed months, Product A undergoes several in-and decreasing developments, culminating in a minimum coverage value in October. This demonstrates that, even though the numeric values might not exhibit absolute comparability, general trends and behaviors can still be captured in a relative comparison.

**Comparability Factor**
It became apparent that the retrieved comparability value can not reflect strong osculations or changes, leading to two important conclusions:

- The derived comparability factor cannot – unless further equipped – reflect extreme changes in the data set.
- Smooth, steady transitions however can be reflected (see figure 24) properly.

The devised comparability factors can constitute an apt "quick-fix" to integrate new data points dynamically during the software development process. Its problem with oscillating values can arguably limit the correct insertion of new values; however, one can counter argue that rapid changes in metrics can always be accounted to limited periods in the software development process, and thus do not constitute the norm.

**Indicator**

In order to arrive at an abstracted indicator value, a calculation method was devised which presents a cumulative, gearing-factor adjusted sum of the provided metrics. The resulting value constitutes an abstracted point of reference which can serve as a means contextualize progress and deviations. The results proved a slight imbalance of values, with Product A exhibiting an indicator of 225.76 and Product B 319.77, graphically reprocess in figure 22).

**Analysis Models - Predictive Models**

74
Each data set underwent regression analysis (linear regression, quadratic regression, cubic regression) to determine which of the models is most parsimonious with the given data. The results underlined the established assumption that it can be sensible to investigate more regression models, as one model might not fit every data set. Given that the complexity metrics decrease considerably over the months under observation, a linear regression model did not capture this behavior correctly. By contrast, however, quadratic and cubic models more closely fit the data’s behavior.

The results presented above constitute valuable insights into the provided data sets, and demonstrate that a more intimate inspection can be beneficial for choosing an appropriate predictive model. By implication, this can have positive influences on the steps of the HOLLY model, as the appropriate analysis influences the better adaptation of the model to new data.

8.3 Limitations

The HOLLY model provides a *theoretical* framework for analyzing software development tailored to multi-language programming environments. Its correct deployment goes hand-in-hand with the proper analysis of the programming languages in question, leaving room for adaptations in the concrete use-case.

The case study conducted in this thesis – and thus the evaluation of the model – was limited by the provided data. A closer inspection of the data sets revealed missing data units, which had to be excluded. Moreover, the cross-language comparativeness was hampered by the fact that different time spans of the metrics in questions were provided, thus reducing the overlapping timespan of the two product lines to 8 months.

Moreover, the short timespan also proved difficult for the proper evaluation of a comparability factor, as this computation will definitely benefit from a more diverse data distribution. The relatively short timespan also has severe implications for the outcomes of the predictive models, as again more data points will yield more robust regression models.

8.4 Outlook and further research

The integration of more programming languages is undoubtedly the most interesting aspect for future research, as it will demonstrate the model’s comparative
expressiveness. Moreover, it would be interesting to compare programming language that do not fall under the same programming paradigm. The comparison of object-oriented and for instance functional programming languages can further highlight the power of selecting an appropriate gearing factor.

Moreover, the integration of process and product metrics will provide a more diverse and rounded picture of the software development process. Especially the inclusion of agile metrics can provide interesting insights into the software development process, even more so when comparing the dynamics of different programming language.

To ensure more dynamic analysis of the data, the tool presented in chapter 6 can be extended to execute all steps of the HOLLY model in a fully automated manner. Also, the addition of more analysis models can provide further useful insights into multi-language software development processes.
9. Conclusion

The proper selection, deployment and analysis of software metrics is exacerbated if more than one programming language and/or software development process is involved. Indeed, cross-language or cross-development comparability can be considered an additional dimension in the deployment of software metrics. Achieving metric comparability, however, depends on the projects under observation and must therefore be tailored to the idiosyncrasies of the software product(s) in question.

The HOLLY model was devised to present an uniform approach for the selection, reprocessing and comparison of software metrics in multi-language environments. In contrast to most research which compares programming languages from a purely numerical point of view, it was this thesis’ aim to understand how the differences between programming languages can be actively incorporated into analysis processes to generate more adaptive comparability models.

The assessment of the HOLLY model showed that the *perspective of interpretation* is crucial for the validity of the outcomes. To receive an overview of the status of two programming languages or product lines, the derived indicators are an apt means; however, only the subsequent observation and interpretation of the underlying metrics enable the induction of suitable countermeasures. The need for interpretation (i.e. comparing two metrics directly) versus the need for abstraction (i.e. an abstract indicator value) here exhibit a continuous interdependency, which must be considered in the evaluation of the provided data.

The integration of more programming languages is undoubtedly the most interesting aspect for future research, as it will demonstrate the model’s comparative expressiveness. Moreover, it would be interesting to compare programming languages that do not fall under the same programming paradigm. The comparison of object-oriented and for instance functional programming languages can further highlight model’s innate cross-language comparability potential.
References


