MAGISTERARBEIT

Evaluating Object-Oriented Software Metrics for Source Code Change Analysis – A Study on Open Source Projects

zur Erlangung des akademischen Grades
Magister
(Mag. rer. soc. oec.)

ausgeführt am
Institut für Rechnergestützte Automation
Forschungsgruppe Industrial Software

der Technischen Universität Wien

unter der Anleitung von
Univ.-Prof. Dipl.-Ing. Dr. techn. Thomas Grechenig und
Projektass. Dipl.-Ing. Bakk.techn. Mario Bernhart

durch
Andreas Mauczka

Burggasse 89/1/2, 1070 Wien

Wien, 02. April 2008

Eidesstattliche Erklärung

Ich erkläre an Eides statt, daß ich die vorliegende Arbeit selbständig und ohne fremde Hilfe verfaßt, andere als die angegebenen Quellen nicht benützt und die den benutzten Quellen wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Wien, am ---

Name

Danksagung

An erster Stelle möchte ich meinen Eltern für ihre Unterstützung danken, die es mir ermöglichte ein Studium frei jeglichen Leistungsdruckes zu absolvieren.

Weiters meinem Bruder und meinen Freunden Christoph, Isabella, Kathi, Markus, Philipp, Thomas, und allen anderen, zu denen ich mit Sorgen und Problemen im Zuge meiner Diplomarbeit gekommen bin.

Kurzfassung

Ziel dieser Arbeit ist es den Bereich der Software-Metriken, konkreter den Bereich der Validierung von Software-Metriken, um einen neuen Ansatz zu erweitern. Die gängige Vorgehensweise zur Validierung von Software Metriken besteht darin, die Werte, die von den Metriken generiert wurden, Fehler-Daten in einer statistischen Analyse gegenüber zu stellen. Die Vorgehensweise in dieser Arbeit widerspricht diesem Ansatz insofern, als dass die Werte der Metriken dem sogenannten Wartungsaufwand (Maintenance Effort) gegenüber gestellt werden. Dieser Wartungsaufwand kann in der Form von Changed Lines gemessen werden und in automatisierter Form gesammelt werden. Dies ermöglicht eine ausreichende Stichprobengröße, die wiederum zuverlässiger Schlüsse ziehen lässt. Änderungen am Source Code sind meistens risikobehaftet und teuer. Änderungsdaten stellen daher einen interessanten Aspekt des Qualitätsmanagements dar. Ein Ziel dieser Arbeit ist es daher ein Model vorzustellen, das in der Lage ist, unter der Zuhilfenahme von Software Metriken, Änderungen aufzuzeigen

Um eine ausreichend große Anzahl an Projekten analysieren zu können, werden Open Source Projekte als Datenquelle benutzt. In der Studie werden zwei Metriken-Suiten zur Bewertung der Projekte hinzugezogen. Mittels selbst-geschriebener Parser werden die Daten für die statistische Analyse aufbereitet. Die Validierung selbst erfolgt durch Anwendung der Regressionsanalyse. Changed Lines und die Summe aus Non-Commenting Source Statements (NCSS) und Changed Lines bilden die abhängigen Variablen. Die von den Metriken erzeugten Werte stellen die unabhängigen Variablen dar.

Ein Set von Hypothesen, basierend auf Annahmen und Konzepten aus der einschlägigen Literatur, wird den Ergebnissen der Regressionsanalyse gegenüber gestellt. Das Ergebnis zeigt, dass je „anspruchsvoller" eine Metrik ist, d.h. je komplexer sie ist, desto geringer ist ihre Aussagekraft als Prädiktor für die abhängigen Variablen. Zusätzlich zu dieser Erkenntnis wird ein Satz von Metriken vorgestellt, der mehr als 60% der Varianz in einer der abhängigen Variablen belegt. Außerdem wird ein zweiter Satz von Metriken vorgestellt, der sehr früh im Software Entwicklungszyklus anwendbar ist, aber dennoch bereits eine Abdeckung von 38,9% der Varianz in einer der abhängigen Variablen vorweisen kann.

Abstract

This study aims to provide additional insight into the area of Software-Metric Validation by using a new methodology based on code change analysis. As code changes are risky and can be expensive, change data provides an interesting background for Quality Management. One of the goals of this work is to provide a model to be able to pin-point change occurrence by using Software Metrics.

The common methodology to validate Software Metrics is by comparing failure data and the metric-generated value in statistical analysis. The methodology chosen for this work is to validate Software Metrics by using change data instead of failure data. Change data can be gathered in an automated fashion, therefore allowing for a big sample size, which in turn delivers more cogent conclusions. This change data is referred to as maintenance effort in literature.

To be able to access a sufficient sample for statistical analysis, Open Source projects are used to gather the change data. Two metric suites are employed to gather the metric values. Self-written parsers are used to pre-process the data for statistical analysis. The statistical method used for validation is regression analysis. Two dependent variables are used, Changed Lines and a sum of Non-Commenting Source Statements – abbreviated NCSS - and Changed Lines. The metric generated values are used as independent variables.

A set of Hypotheses, based on assumptions and concepts in literature, is pitted against the results of the regression analysis. The results show that the more “sophisticated” a metric, i.e. metrics that are more complex than others, the less suited it is to predict the dependent variables. The possible reasons for this are discussed in this work. Furthermore a set of metrics is found that serves as predictor for more than 60% of the variance in one of the dependent variables. Also, a set of metrics that can be employed early in the development cycles and that still delivers 38,9% coverage, is introduced.

Index

Index of Figures

Index of Tables

1 Introduction

1.1 General

With the growing importance of quality management in software industry since the early 90's, several means to measure or improve software quality have been tried and tested. This work will examine a certain aspect of measurement of software quality. Before this aspect can be discussed though, a definition of quality that will be used in this work is required to avoid confusing terms. The definition of quality itself is hard to nail down as it is dependent on the role or view that is defining it. A customer's view of quality in a product is of perceivable flaws, while a software engineer might define quality of a product as a factor of the maintainability and reusability of components.

Quality in this work is therefore defined as consisting of three parts: product quality, process quality and customer satisfaction. Process quality and customer satisfaction are both important aspects of quality; however this works focus is on product quality. The reasons why product quality was chosen will become apparent in the course of this chapter.

The first step of improvement is to find some sort of measurements or scales for the items of interest. If it cannot be measured, it is impossible to prove any form of enhancement. This measurement of quality is done with metrics. While there are metrics for process quality and customer satisfaction, all metrics examined later on will concern themselves with product quality.

Product quality for software is often measured in two most basic ways: defect rates (defects per size-unit e.g. lines of code) and reliability (failures per time-unit) [KaSt02]. These measures are metrics themselves (as defined earlier). They represent the most basic measures of software quality and are therefore often used to validate more complex metrics.

Since the measurements of defect rate and reliability are failure-dependent and simple measures, other software metrics were validated by comparing their performance and predictions to actual defect rates ([BaBr96] [KaCa85], [HeKa81], to mention a few). This work will introduce a different base metric for software quality based on the assumption that in the end a failure is something that increases overall effort. The higher the quality of software the less effort is required to maintain it. This effort will be expressed in changed lines of code per unit.

The structure of this work is based on a standard data-mining process. This means that the work follows a strict pattern to gain knowledge. First a database is determined, and then a dataset is selected according to certain criteria. After these steps are finished, the data is pre-processed to achieve a uniform format of our data on which the next step, the actual mining, can be performed [MeSu99]. Out of the netted information, knowledge will be gathered and used to discard or validate the hypotheses that will be introduced in this work.

1.2 Motivation

The motivation for this work is to provide a different angle on validating software metrics than the “industry standard”. There is, as indicated in the introduction, a lot of research describing the validation of software metrics based on error-data. The material on maintainability to validate metrics is rare and only one publication on object-oriented metric validation was found during research ([LiHe93]). Both kind of studies (error and maintainability based approaches) lack sufficient statistical data. [LiHe93] offers two projects for validation, which is not enough to propose a general validation. So one motivation for this work is the novel approach, the second is to actually evaluate the metrics against a significant number of projects.

Another motivation for this paper is to shed some light in similarity of designs and “styles” of open source projects. In the area of open source projects, a lot of research is done to analyze the structure and working mechanisms of open source development teams. This work will provide insights that can be further used to describe the fascinating open source phenomena.

1.3 Goals

The goal of this work is to evaluate the following three hypotheses (similar to [NaBa06]):

Hypothesis 1: A high value of Changed Lines in a project will lead to a significant score of the metric under examination.
Hypothesis 2: If Hypothesis 1 holds true, is there a set of metrics that is valid in all projects under examination?
Hypothesis 3: Is this set of metrics viable on other open source projects?

Hypothesis 1 will be evaluated against approximately 100 (selected based on predefined selection criteria) projects on SourceForge. Studies undertaken in the field of error-data and validation of metrics indicate that there will be a set of metrics that will hold true (in determined boundaries) for the complete sample. However these studies where done on a small number of projects, so the outcome of this work may be different. Based on the different approach of this work, the result is expected to vary when it comes to which metric will be “successful” and which won’t. Furthermore the used programming language, namely Java, is different to the only similar study [LiHe93] in the field.

This work aims to anticipate which parts of a project will require more effort in an early stage of development based on applying the metrics. SourceForge projects could be analyzed at different stages of development to evaluate the performance of the metrics set constructed in Hypotheses 2. This evaluation itself will be discussed in future studies; this work aims to deliver a solid base for these researches.

1.4 Structure

Chapter 2 will give a short introduction into object-oriented metrics in general and code and structure metrics in detail.

Chapter 3 deals with the first two steps of the data-mining process, Selecting and Pre-Processing the Data. In this chapter it will be analyzed where the data used in this work comes from and which criteria were chosen for selecting the used sample. Then the way the data has been formatted and transformed to be suitable for the parsers and metrics will be explained.

Chapter 4 will discuss the tools used to gather the metrics and how they implemented the metrics introduced in Chapter 3. Each metric that will be used later on for evaluation will be explained here in great depth.

Chapter 5 picks the data-mining-process up where it was left in Chapter 3 and deals some more with the pre-processing of the CVS data. This chapter explains the tools programmed for this work, however the tools come into use at different parts of the data-mining-process.

Chapter 6 will present the mined results of the metrics. There will be results for each metric and an overall result. This chapter represents the mining step.

Chapter 7 tries to derive knowledge out of the information delivered in Chapter 6 and tries to postulate a set of metrics that is able to predict maintainability in modules.

2 Object-Oriented Metrics

2.1 General

Before object-oriented metrics can be discussed, metrics themselves need to be defined. As mentioned in the introduction, metrics provide the means to measure software quality. Metrics are available for each kind of quality (product, process and customer satisfaction), but for this work only product metrics are of interest. Some of the most basic metrics are defect-rates and mean-time-to-failure (reliability). The terms failure and defects can be used interchangeably [KaSt02]. Another measurement for quality (a very simple metric) will be introduced in this work and the more complex metrics will be evaluated against it. This metric will be changed lines per unit (a unit can be e.g. a class or a package).

There are several definitions of structure and code metrics, and several different ways to classify metrics. In [HeSe96] metrics are categorized by what they are measuring, either external or internal characteristics. External characteristics are quality factors, defined as maintainability, reusability, abstraction, usability, availability and reliability. Internal characteristics are objective measures like size, control flow complexity, inter-module coupling and modular cohesion. Then they are defined by how they measure. Product metrics are defined as snapshots of a certain point in time, while process metrics are defined as measures over time.

These two definitions vary greatly with other literature on this topic so there is need to define some terms for this paper. While [KaSt02] defines defect-rates and mean-time-to-failure as product quality metrics, [HeSe96] defines these as classic process metrics as they are measuring over time. The metrics used in this paper are product metrics in the sense of [HeSe96]; this means that they are indeed snapshots of the projects at the time of the CVS download. The changed lines metric is a process metric per definition of [HeSe96], since it measures all changed lines over the whole project lifetime. However the parallels to [KaSt02] allow this metric also to be categorized as a product metric.

After settling for product metrics as broader definition for the employed metrics, the next step is to detail the kind of metrics which will be used in the field. The metrics that will be evaluated are of two out of three kinds, structure, code and hybrid metrics. A structure metric is a metric that assigns a value to certain design aspects of a project. A code metric is a metric that analyzes the source code itself and assigns a value to certain design aspects of a code-unit. Structure metrics deal with the relationships between components, while code metrics concern themselves with the components themselves. There is a third category named hybrid metrics, which is used for the metrics combining both features (e.g. information flow weighted by lines-of-code) [KaDe85]. Hybrid metrics are not used in this work.

The next term that needs definition for this work is the term of complexity. Some software metrics measure quality in the context of complexity. Complexity is broadly defined and a metric measures only aspects of complexity. While e.g. McCabe’s cyclomatic complexity measures the complexity of a module, it does not measure the complexity of the interplay of the modules of a system. Therefore McCabe’s metric measures procedural complexity while the coupling-between-objects (CBO) metric measures design complexity. According to this definition the structure metrics used in this work measure system design complexity, while the code metrics measure procedural complexity. These definitions are based on [HeSe96], with the difference that in [HeSe96] the metrics are categorized by the phase in the development process in which they can be employed. E.g. one would assume that Depth of Inheritance would be a metric to measure system design complexity, but since the metric is run on the code it is classified as a metric that measures procedural complexity in [HeSe96]. This seemed confusing and therefore this work will just use the terms structure metrics for metrics that measure the interplay of components (Inheritance, coupling and abstraction) and code metrics for metrics that measure the components themselves.

The metrics used in this study have been adapted or designed for the object-oriented paradigm.

2.2 Changed Lines Metric

The goal of this study is to show a correlation between changed lines of a unit (be it package or class) and the score of a metric. There has been criticism of using only one dependent variable (the changed lines) in [KaCa85], based on the fact that one variable will introduce too much of a bias, e.g. the style of the programmer has too much of an impact on lines of code. This might hold true for the size of the sample analyzed in [KaCa85] where three projects are analyzed. This is very unlikely to hold true for the size of the sample analyzed in this work. While this work uses up to three dependent variables at some point, the dependent variables are all derivates of the changed lines metric and are not different dependent variables in the sense of [KaCa85].

The term changed lines as is used in this work means all added and deleted lines recorded in the CVS repository and summed up. Since changes themselves are not offered as data by the repositories, the sum has to be used. The reasoning for using the sum is that the number of undertaken changes would have to be calculated by heuristics (like [BoGu06]). While there is a lot of research done in this area, the goal of this work is to provide some first insight into validation of software metrics by changed lines. Using these heuristics would pan the study into the direction of the heuristic used and introduce bias into the study. By adhering to the most basic value of the sum of additions and deletes, the possible bias is limited to the commit-pattern of the programmers (e.g. the number of commits of a single file). This is averaged out by the size of the sample.

During the preparation process for the study, normalisation of the changed lines sum via unit size was considered and then dropped because of the following reason. If class C1 contains 100 lines and 10 methods and has 200 changed lines and class C2 contains 50 lines and 5 methods and has 100 changed lines, we would have e.g. a Number Of Methods (a metric) score of C1 = 10 and C2 = 5. The normalized factor of changed lines/size would be 2 in both cases. This would make the changed lines obsolete as the metric in this example is a size metric. Normalizing the dependent variable via size would just render the variable of any insight in the relationship to most metrics. However size does play a part in calculating the changed lines, as the size of a class is a one-time addition, which can be interpreted as a one-time change. The data will be analyzed with two variations of changed lines measurement to reflect this.

The standard for validation of metrics are, as mentioned earlier, error based measures. This new way of changed lines should allow for analyzing larger numbers of projects compared to what has been done so far in this field ([BaBr96] [KaCa85], [HeKa81 and [LiHe93]). The analyzed projects in earlier validations were limited because of the nature of error data. E.g. Error data for SourceForge-projects is handled completely different; some projects use their own versions of error data trackers, some import their previous error data, etc. It just introduces too many liabilities to be able to analyze a suitable set of projects. The measure used in this work is not without influences either but has significantly less. It is influenced by the usage of the CVS system and by programming style. This is evened out by the large sample.

The last issue that needs to be explained in the changed lines measure is the fact that the sum of added and deleted lines is used. One could assume that if a log contains the same numbers of added and deleted lines and the diff shows similar code that a change was done. This work concerns itself with the changed lines at the current stage of the project though. It does not matter if there was one huge change or several small changes, neither does it matter if lines were changed or added and deleted. The goal is to find out that additional effort in form of adding, deleting or changing has been introduced at a certain unit in the system. To interpret this effort is not the goal of this work. This work aims to see whether these changes can be used to predict future effort (again, this effort is not interpreted) in modules. To put a value on the effort predicted is ground for future studies. Here heuristics can be used to determine change dimension and type. These heuristics interpret the characteristic of the measure used here, while this work interprets the value of the measure against metrics.

2.3 Structure Metrics

Structure metrics deal with the coupling of components. Similar to complexity, coupling requires a definition as it is broadly and ambiguously used at the moment. A coarse definition is that coupling describes the relationship between objects in the object-oriented paradigm. There are however different kinds of coupling. There is intermodule coupling, described in [HeSe96] (taken from [LoM93]). It is *"measured by the number of relationships between classes or between subsystems" - [HeSe96]* (taken from [LoM93]). This is a helpful generalisation, but not finely grained enough to work with. In [LiHe93] there are three kinds of coupling described, though only two of interest for this paper. One is coupling through inheritance, which is coupling between a class and its superclass in a hereditary manner. The other is coupling through message passing, which is coupling of two classes by communication.

This distinction is important as coupling metrics sometimes measure both, sometimes only one of these. The coupling metrics used in this work will be introduced in Chapter 4 and it will be explicitly stated which kind of coupling they are measuring.

After defining what is measured by structure metrics the implications of the measurement need to be discussed. [MaRo03] gives the following definitions for designs according to their relationship patterns. The first term introduced is stability, which is defined as *"… is related to the amount of work required to make a change" – [MaRo03]*. This definition is then put into context with complexity by claiming that a package which has many other packages depend on it (therefore increasing design complexity) increases the difficulty of changing it, because all the consequences for the depending packages have to be considered. Therefore a package that has many packages depending on it is defined as stable. If a package does not depend on any other packages it is called "independent". If a package has no other packages depending on it is defined as "irresponsible" [MaRo03].

Stability therefore should indicate components that change less, while instable components are expected to change more. An instable component does not imply bad design. If all classes were maximally stable, the system would be unchangeable, so an amount of stable and instable units is required [MaRo03].

Inheritance and Abstractness metrics also fall under the definition of structure metrics in this work. They are mentioned in the package design chapter in [MaRo03]. However inheritance was excluded from design complexity in [HeSe96], where they are put in the procedural metric section. This work defines inheritance as part of the system design and therefore are metrics measuring inheritance part of the structure metrics. Abstractness is defined as part of the structure metrics as well as it is an aspect of the system design.

2.4 Code Metrics

Code metrics in this work are defined as metrics that measure the procedural complexity of one unit of code (e.g. a Class) (Refer to the definition given in Section 2.1). Procedural complexity includes size, data structure and logic structure. The definition of code metrics used here is similar to module metrics (metrics that measure procedural complexity) in [ChKe91]. Module metrics consist of style metrics, size metrics, data structure metrics, logic structure metrics and internal cohesion metrics.

The first kind of module metrics to explain are size metrics. Due to the changes of the programming paradigm from functional to object-oriented, size measures that are accurate needed to be designed. Simple lines of code counts proved to be deficient because of inheritance (there’s mention of a study proving to reduce code by 80% when using the object-oriented paradigm in [ChKe91]). Therefore methods like the Number-of-Classes metric were designed. Inheritance and Abstractness are not accounted as size metrics (see Section 2.3). Size metrics measure the dimensions of an object-oriented program in some form.

The next metrics are logic structure metrics. These metrics measure the internal logic of a unit, e.g. Number of Decisions inside a class. The most popular metric in this area is McCabe’s cyclomatic complexity, which will be explained in detail in Chapter 4.

Data structure metrics and style metrics will not be discussed in this work. Data structure metrics concern themselves with the scope of variables (like number of variables, is the variable referenced once, etc.), while style metrics analyse items like indention, comments in the code and similar patterns.

Cohesion describes the grouping of related tasks in software units (e.g. methods of a class). Therefore, internal cohesion metrics in object-oriented systems measure the “... *degree of similarity of methods” – [ChKe91]* or the lack of it. If two methods are similar, it is assumed that they will perform related tasks. In [ChKe91], similarity is defined by the number of shared instance variables of methods. The measured lack of similarity is called Lack Of Cohesion Of Methods (LCOM) in literature. There are several methods that measure LCOM, the variant used in this work will be described in the metric suite chapter (Chapter 4) in depth.

3 Selecting and Pre-Processing the Data

3.1 General

This chapter deals with the first two steps of the data-mining process. It explains which database will be used in the study and the reasoning behind the decision process. Furthermore, the chapter provides the arguments as of why the final sample in the study has been selected and based on which criteria. The first part of the chapter gives a short overview of SourceForge [SoFo] and the project statistics available. The goal is to provide the reasons behind the selection process, which is then performed in the second part of the chapter. The final part of the chapter describes how the data was prepared for the metric tools employed in the later chapters of this work.

3.2 OSSmole

SourceForge contains a vast amount of free, libre or open source software projects (FLOSS). Due to the nature of open source projects, the majority of these projects is either done by individuals and/or were started but never finished. Furthermore Source-Forge contains projects in different programming languages. To obtain the best possible results for our research, it became obvious that a set of sound parameters must be obtained. To be able to find these parameters statistics on SourceForge projects would need to be assessed. These statistics can be gathered by three means:

- Spidering the SourceForge-Project pages ourselves
- Using the data of someone else that already spidered SourceForge
- Obtaining the project data directly from SourceForge

Option one was discarded swiftly on the basis that the amount of time required to spider SourceForge would be tremendous and the effort, since it is just a pre-selection criterion for the analysis later on, would not be justifiable by any means. The third option was tempting and SourceForge does indeed offer dumps of their project data-base for educational or scientific purposes. To get these dumps, an application to access the data needs to be filed at the University of Notre Dame with the specific project data and then access to the data is granted [NoDa]. Needless to say the data available is huge and again was discarded as too much effort for the granulation of data that would be required. So option two was selected. Early in the research process the OSSmole-project was discovered and after abandoning the options mentioned afore it was found best suited for the data needed. The access is simple, it is up-to-date and it is completely open for access.

„OSSmole is a collaborative project designed to gather, share and store comparable data and analyses of free and open source software development for academic research.

The project draws on the ongoing collection and analysis efforts of many research groups, reducing duplication, and promoting compatibility both across sources of online FLOSS data and across research groups and analyses.“ – [CoHo05]

Though OSSmole offers dumps on several “forges” (Freshmeat, Rubyforge, etc.), SourceForge was chosen as it is the largest of the ones available. Diversity of projects was achieved by not limiting the projects to the same genre, even though only one “Forge” was used. It is doubtful that the nature of Open Source Projects will vary greatly between the “Forges”, so only one main-source of data was chosen. OSSmole spiders the SourceForge-project pages and offers all the data available there as an SQL-dump. The data is updated on a monthly base. This work uses the SQL dump of August 2007 for its analysis.

The SourceForge data set available at OSSmole consists of the following Data Elements (Source: [OsSo]):

Project Items

- Project names (long name *and* short unique 'unixname')
- Project Descriptions
- Project URLs (URL on Sourceforge and 'real' URL)
- Project registration date
- Project intended audience(s)
- Project license(s)
- Project programming language(s)
- Project database environment(s)
- Project operating system(s)
- Project donor(s)
- Project status (alpha, beta, mature, etc)
- Project topic(s)
- Project user interface(s)

Developer items

- Project developers (username, real name, Sourceforge email address)
- Developer role(s) on project, including whether an administrator or not

Statistics items

- Project downloads (sum of project downloads over 60-day window)
- Project ranks (project rank averaged over 60-day window)
- Project tracker sums (sums of tracker opens and closes over 60-day window)

These data elements represent all information that can be gathered from the project sites of SourceForge. SourceForge does not offer lines of code or any other form of code statistics in a centralized fashion. To provide source code statistics one would have to access all CVS repositories. OSSmole was therefore selected to provide the data required for the next step of our data-mining process. Another tool to access source code information was chosen additionally.

3.3 Finding the Parameters

To find suitable parameters to determine the final projects and to pre-process the data from OSSmole, an overview of the available data at SourceForge was needed. For a short glimpse of the data, before venturing ahead in an SQL dump, [WeDa05] was used. The parameters should decrease the whole set of SourceForge-projects to about 300 (the aim was 100, but this had to be increased due to reasons mentioned later on) and should provide data that is diverse but significant. This means the projects should be big, but not huge, in a mature and stable part of their development process and show a certain activity. It was necessary to provide an overview of SourceForge projects to find the right values for the selection criteria.

SourceForge offers categories for the maturity of their projects. These categories are:

Phase	**Code in Database**	**No. of Projects**
Beta	10	26435
Production/Stable	**11**	**22176**
Planning	7	21828
Alpha	9	19670
Pre-Alpha	8	17773
Inactive	358	2931
Mature	**12**	**1914**

Table 1: SourceForge Projects by Status, August ‘07

The emboldened rows show the projects of interest for this work. These projects are almost completely developed or at least at a point of development where drastic changes are not very likely. They were chosen because they should be able to offer solid changed lines, as all the major changes should have been performed somewhere in the history of the projects. 24090 projects are in the categories “Mature” and “Production/Stable”.

Since it would prove impossible to analyze projects in all programming languages, a language had to be chosen.

Description	Code in Database	No. of Projects
Java	198	25656
C++	165	21849
C	164	19242

Table 2: SourceForge Projects by Language, August '07

The topic of this work is object-oriented metrics, so an object-oriented language was opted for. Java was favoured over C++ because of the large number of tools available in the metric sector for Java. The reason that this study was not done on projects of different programming languages was the effort of developing a parser for each language and metric suite. There are currently 25656 Java projects on SourceForge. On a note worth mentioning, not all of these projects actually are Java projects. A lot of projects have been flagged erroneously. E.g. a project with 40.000 lines of Python code and 1000 lines of Java code is flagged as Java project. This is one of the reasons the 100 final projects had to be handpicked out of the set goal of 300 projects.

The next criterion is the number of developers working on a project.

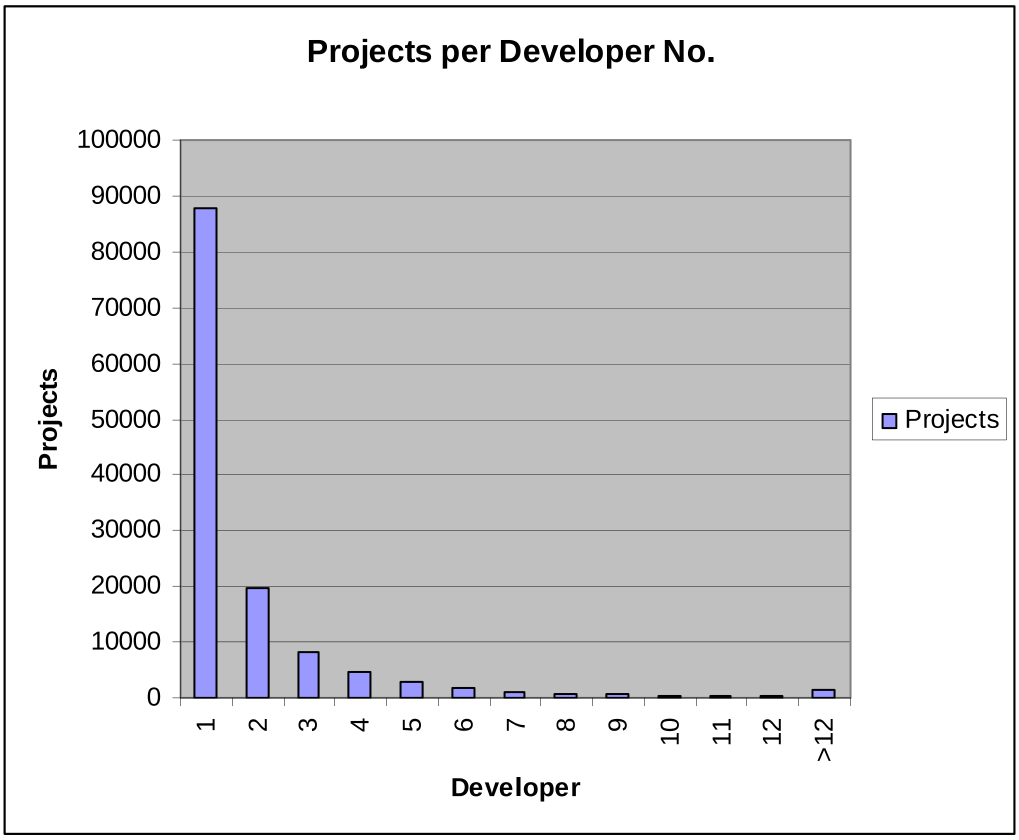

Figure 1: Projects per Developer No., August ‘07

As can be seen in Figure 1 the main portion of Projects is developed by a single developer. The average team size is ~2 (1,9725) developers per team and the standard deviation is 3,5. There are 1436 projects with more than 12 developers. This trend to small scale projects for SourceForge makes for a hard selection criterion based on team size.

The next item of interest for this work is the lines of code per project. Unfortunately SourceForge does not offer statistics for this, so another mean to find the lines of code had to be resorted to. The solution was found in Krugle, a source code search engine that holds all SourceForge projects [SoKu].

The next criterion for our selection is the ranking system of SourceForge. The aim is to get active and popular projects, i.e. projects that are downloaded regularly and are reviewed often. The reasoning is to capture projects that are in similar final stages of development. The formula for the SourceForge ranking system is as follows (Source: [SoDo]):

```
Traffic:
   (
    (log(prior 7 days download total + 1) / log(highest all-project
download
total + 1))
   +(log(prior 7 days logo hits total + 1) / log(highest all-project
logo hits + 1))
   +(log(prior 7 days site hits total + 1) / log(highest all-project
site hits + 1))
   ) / 3

Development:
   (
    (log(prior 7 days cvs commit total + 1) / log(highest all-project
total + 1))
   +( (100-age of latest file release (in days, max 100)) / 100 )
   +( (100-days since last project administrator login (max 100)) /
100 )
   ) / 3

Communication:
   (
    (log(prior 7 days Tracker submission count + 1) / log(highest all-
project total + 1))
   +(log(prior 7 days ML post count + 1) / log(highest all-project to-
tal + 1))
   +(log(prior 7 days Forum post count + 1) / log(highest all-project
total + 1))
   ) / 3


total = traffic + development + communication
```

The activity ranking of SourceForge consists of three aspects, traffic, development and communication. Each aspect has three subcategories. It is not only an activity value, but also a ranking system, weighted against the maximum value of a project in each subcategory.

3.4 Querying the SQL Dump

After reviewing the characteristics of interest on SourceForge, the next step is to find the boundaries to limit the projects. The latest SQL dump was imported from OSSmole (August ’07) and a MySql ([MySq]) database was set up. SQL queries were performed to reduce the set of projects to a solid sample.

The first step was to decrease the pool of all projects on SourceForge to the ones at production/stable and/or mature level. This left 24090 projects as shown in Table 1. Note that there are a few double entries as some projects have actually two categories (production/stable AND mature).

Since this work focuses on Java projects, all other projects were discarded. This reduced the number of projects to 5313.

The next part to furthermore reduce the number of projects was to find out lower and upper boundaries for the actual project size. Since Eclipse will be a prerequisite of at least one metric suite employed, the projects should not be too large. Since the capacities for running automated analyses will be limited as well, a few tests were run and the upper boundary of around 120000 lines of code was found. These tests were basically trial and error based, so that a project was randomly chosen and seen if it could be worked on in a sensible amount of time. 30000 lines of code was chosen as the lower boundary, as anything below that threshold would not be significant in the structure metrics (structure metrics use concepts that are not necessarily implemented in small scale projects). Since SourceForge does not offer the lines of code of a project, the Krugle Search Engine on SourceForge was used to find out the lines of code of the chosen projects. However this was only viable as last step in the selection process since it had to be done manually. So a way to predict lines of code of a project had to be found.

To predict project sizes team size was tried as a parameter. Since Figure 1 shows that there are a lot of small (1 or 2 developers) sized teams, the average team size was out of the question as a base for selecting projects. To find out viable developer boundaries, a random sample of the projects was necessary. To define the first boundary, twenty projects between 13 and 18 developers were chosen at random. Note that the twenty projects were chosen from the pool of Java projects at production/stable or maturity level and that any projects with a mix of programming languages were discarded if the mix had more lines of code than the Java part. The average lines of code for this sample were 126195 and out of the boundary introduced earlier. Ten projects with 12 developers were examined and had an average of 87315 lines of code, which was deemed a good value for the boundary aim of 120000. Unfortunately the standard deviation for the sample was at 81304. Still 12 developers was chosen as upper boundary since the final projects will have to be handpicked anyways based on their lines of code after running the other queries on the projects, so it just helps to tighten the pool of projects further.

The next step was to find the lower boundary for the team size. Teams of the sizes 1 or 2 were discarded immediately due to their overrepresentation and that it was believed that teams of that size would not produce enough code to run the metrics on. A sample of 20 projects was taken with a developer size of 6 and produced an average of ~65000 lines and a standard deviation of ~109000. Since it is doubtful that it will be possible to get better values for a lower threshold, 6 developers per team was chosen.

The high variations in both samples can be explained by the low numbers of the sample and because the projects have not been filtered by activity yet. The reason is that projects with a high number of lines of code (one of the projects in the 6-developer-sample had 490000 lines of code) maybe had a huge number of developers at a certain point in their development process, nowadays the numbers of developers have dwindled though. Another reason might be copy-and-pasted projects that started under different surroundings. All of these reasons just emphasize the need to pick the final projects carefully and manually.

The boundaries of 6 to 12 developers per project resulted in a pool of 428 candidates.

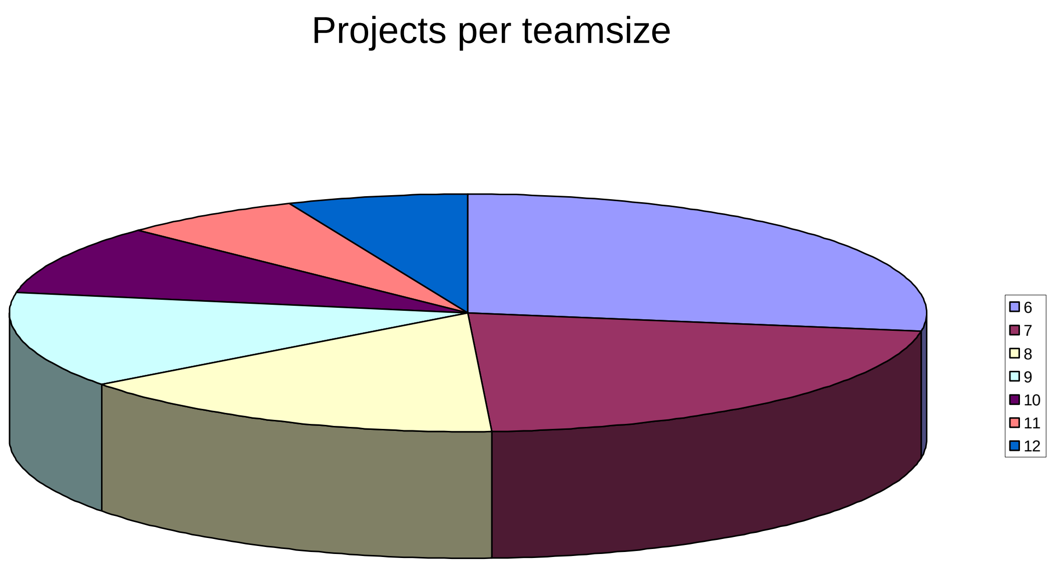

Figure 2: Projects per teamsize in the final sample

The next step was undertaken to decrease the projects to around 300 (so it would be possible to handpick 100 projects for our study). This was achieved by using the activity ranking of the projects and picking the first 300. Out of these 300 projects about 100 projects were selected for the study.

(Data gathered from [OsSo])

3.5 Acquiring the CVS Data

The next step after the selection of our database is to gather the data and to pre-process it so later on mining via the metrics can be performed. A small tool was developed to generate a script that automatically downloads all CVS data from a list of projects. This tool will be explained in further detail in Chapter 5. After the projects are made available locally, they are imported into the Eclipse workspace. In the next step the project file from Eclipse is edited to change the builder to the metrics builder. This step is necessary so the project does not have to be rebuilt completely.

After refreshing the project, it is ready for examination via the metric suite. This step will be explained in further detail in Chapter 5.

The second part for the analysis stems from the CVS history of the projects. This is acquired by running the CVS-log command on the repositories and saving the output into log-files. These log-files can then be parsed and the changed lines for each file in the repository can be found out. The functionality of the employed CVS parsers is explained in Chapter 5 as well.

4 Applied Metric Suites

4.1 General

To gather the data for the evaluation, two suites of metrics have been applied. One metric suite has been used to gather the structure metrics, while the other has been used to gather the code metrics (and one structure metric). Furthermore a tool has been employed to automatically count Non-Commenting Source Statements (NCSS).

The following metrics have been chosen for this study:

Code Metrics

- Weighted Methods per Class (WMC)
- Lack of Cohesion of Methods (LCOM)
- Number of Methods (NOM)

Structure Metrics

- Efferent/Afferent Coupling (EC/AC)
- Depth of Inheritance Tree (DIT)
- Abstractness

The metrics have been chosen because they measure different aspects in their relative resorts. The WMC-metric measures procedural complexity (the complexity of a class), LCOM-metric measures the similarity of methods of a class and the Number of Methods metric is considered a size metric. Efferent and Afferent Coupling metrics measure system complexity from an interaction-based point of view, while the Depth of Inheritance Tree metric measures system complexity from an architectural point of view.

In the following chapters the metrics will be analyzed and the implementation in the metric suites will be discussed.

4.2 JavaNCSS

JavaNCSS ([JaNc]) was used to count the NCSS for the classes and packages. Furthermore it substitutes the NOM-Metric explained in Section 4.4, as the implementation of the NOM-Metric in the Metrics-Plug-In proved to offer values widely varying to the results achieved by JavaNCSS. The values of JavaNCSS were similar to manually counting the metric out in a few test-projects and therefore JavaNCSS was used.

4.3 JDepend

JDepend comes as a standalone program or as a plug-in for Eclipse [ClJd]. For this work, the standalone program was chosen (the reports could have been created from the Eclipse plug-in as well) because of the possiblity to use it from command line and therefore to automate it. JDepend implements the three structure metrics Efferent/Afferent Coupling and Abstractness on a package level according to [MaRo03]. It offers an Instability metric as well, however this metric won't be analyzed in this work. Instability is just Efferent/Afferent Coupling put together into one formula. This poses a problem, as if there is no Efferent Coupling there is always Instability of 0. This work aims to examine the separate effects of Efferent/Afferent Coupling on Source Code Change. The chart for Instability will be added in the Appendix.

Efferent Coupling

Efferent Coupling represents one half of the stability term defined earlier. It represents the number of classes inside a package depending on classes outside of that package [MaRo03]. A high measure of Efferent Coupling therefore suggests an instable package, because it depends highly on classes in different packages. It is very likely to change, as it has many sources for change.

Afferent Coupling

Afferent Coupling represents the number of classes outside of a package depending on classes inside that package [MaRo03]. Afferent Coupling is an indicator for a stable package, i.e. a package that is unlikely to change. Afferent Coupling represents the other half of stability. Afferent and Efferent Coupling can be used to determine the Instability of a package by applying the formula:

$$I = \frac{C_e}{C_a + C_e}$$

If a package does not depend on any other packages it is completely stable. This lack of distinction between a class with high and low Afferent Coupling and no Efferent Coupling renders this formula obsolete for the changed lines metric.

Both metrics have been implemented by counting import statements and qualified names [MaRo03].

Abstractness

"A package should be as abstract as it is stable" – [MaRo03]

The reasoning behind this quote is that stable refers to a unit that has a lot of other units depend on it; in this case a package. This means that if changes have to be made they will be hard to do, because of the impact on the depending classes, so a design is needed that is *".. flexible enough to be extended without requiring modification" – [MaRo03]*. This design is the use of abstract classes. The Abstractness-Metric implemented in JDepend measures the abstractness of a package with a rather simple formula:

$$A = \frac{N_a}{N_c}$$

Abstractness is the number of abstract classes divided by the number of all classes (in the package).

The following hypotheses are being tested against these metrics:

Hypothesis 1.1: A high value of Efferent Coupling of a package will lead to a high value of changed lines. This is based on the assumption that instable packages are changed easier and therefore are changed more often [MaRo03].

Hypothesis 1.2: A high value of Afferent Coupling of a package will lead to a low value of changed lines. According to [MaRo03] a stable package is harder to change and therefore will change less.

Hypothesis 1.3: A high value of Abstractness suggests a stable package. Therefore a high value of Abstractness should lead to a low value of changed lines.

Hypothesis 1.3.1: A high value of Abstractness suggests a stable package and should therefore have a high value of Afferent Coupling. Refer to appendix for chart.

4.4 Metrics Plug-In for Eclipse

The Metrics Plug-In for Eclipse [SoMe] is used to run the remaining metrics on the projects from SourceForge. The Plug-In is run within Eclipse and generates reports as XML-Files which are analyzed and parsed. It implements four metrics that will be used in this work. These are Weighted-Methods-per-Class (WMC), Lack-of-Cohesion-of-Methods (LCOM), Depth-of-Inheritance-Tree (DIT) and Numbers-of-Methods (NOM). All metrics employed by the Plug-In for this work are run on class base. This means that each value delivered by the metric responds to a class. This is different to the JDepend program as it delivers values for packages. To avoid confusion, the DIT-Metric is a structure metric following the definition introduced in this work; however it is employed on class base and not on package base like the other structure metrics.

Weighted Methods per Class

The WMC Metric was introduced in [ChKe91] and broadly defined as

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

c_i stands for the complexity of the method i. The metric used to measure complexity is not defined in [ChKe91]; it is just the groundwork. If all methods have a complexity of 1 WMC equals the NOM metric. The goal of this metric is to measure complexity of a class by adding the complexities of its methods. The metrics reasoning is that number of methods and complexity of methods is an indicator of time and effort [ChKe91]. This is based on the assumption that complexity (for this work McCabe's definition of complexity will be used) introduces time and effort.

McCabe's cyclomatic complexity is based on graph theory. The cyclomatic number of a graph is calculated as:

$$v(G) = e - n + p$$

Where v(G) is the cyclomatic number of Graph G, e is the number of edges, n is the number of nodes and p are the connected components. McCabe further proposes that *"In a strongly connected graph G, the cyclomatic number is equal to the maximum number of linearly independent circuits" – [McTh76]*. A graph is strongly connected if there is a path from any node of the graph to any other node of the graph. A node in this graph represents a block of code that is a sequential flow. The edges of the graph represent the branches a program can take. This graph is known as the program control graph [McTh76]. Note that this graph is a directed graph. To satisfy the definition of a strongly connected graph, the enter and exit points of this program control graph need to be connected.

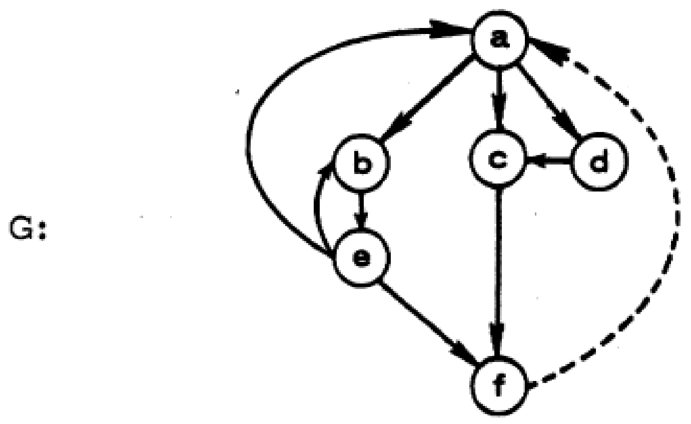

Figure 3: Directed Strongly Connected Graph, Source [McTh76]

The nodes a and f are the enter/exit points and the intersected edge from a to f is the additional edge to make the graph strongly connected. This is reflected by modification of the formula to:

$v(G) = e - n + 2$

For the example presented in [McTh76] and Figure 3, this means that there are five linearly independent circuits. These are *abefa*, *beb*, *abea*, *acfa* and *adcfa*. Using these five basic paths, every path going through the Graph G can be constructed.

The formula used in the Metrics Plug-In is that a counter that starts with v(G) = 1 is incremented by 1 for each *"if, for, while, do, case, catch and the ?: ternary operator, as well as the && and || conditional logic operators in expressions" – [SoMe]*. This is in accordance with the formula presented above (after a brief examination of the metric values on a project).

The WMC value is the summed McCabe's cyclomatic complexity value of a class. For a complete introduction into McCabe's cyclomatic complexity refer to [WaMc96].

Lack of Cohesion of Methods

The LCOM Metric for object-oriented programs was suggested in [ChKe91] and defined as *"the number of disjoint sets formed by the intersection of the n sets" – [ChKe91]*, where n are the sets of instance variables used by n methods (set I_1 belongs to M_1, etc). The formula for similarity σ is:

$$\sigma(M_1, M_2, ..., M_n) = I_1 \cap I_2 ... \cap I_n$$

This formula is taken from [HeSe96], as the formula given in [ChKe91] is erroneous (refer to [HeSe96] for details). σ is the similarity of the Methods M_1 to M_n of a class. If two methods do not share any instance variables, similarity is 0 and they build one disjoint set, ergo LCOM = 1.

The implications of this metric are that cohesiveness of methods is an indicator for well designed classes, as similar tasks are grouped together. If cohesion has a low value it is suggested to break the class up into subclasses. The third implication is that low cohesion increases complexity [ChKe91]. The third implication will be of special interest of this work as it is not clear at all that low cohesion has an impact on complexity at all.

[HeSe96] criticizes this formula for similarity as it is argued that it is very likely that similarity will be zero or small and therefore the formula in this form is useless. A new metric is suggested in [ChKe94], which seems to be a valid measure for LCOM on first glance. LCOM is the number of empty intersections of two sets of instance variables of methods minus the number of non-empty intersections, if the number of empty intersections is bigger than the number of non-empty intersections, otherwise it is 0.

As is pointed out in [HeSe96] this measure fails to deliver an interpretable result. The reasoning is that it does not differentiate enough between classes with high and low cohesion. A new measure for LCOM is suggested in [HeSe96].

A set of m methods accesses a total of a attributes. Perfect cohesion is defined as all methods accessing all attributes. This should deliver a result of 0, while if m = a, so all methods only access a single variable, LCOM* will be 1.

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

Where $\mu(A_j)$ is the number of methods which access each datum. The implementation of LCOM used in the metrics suite is according to this formula. The author of the plug-in cautions the use of LCOM* in Java as using getter and setter methods to access an attribute will indicate a higher lack of cohesion than there really is as only the method with the getters and setters will access the attributes directly while the other methods will access them via the getters and setters. The study in this work provides more insight on this matter.

Depth of Inheritance Tree

This metric measures the depth of a class in the inheritance hierarchy. Following this it is a measure of *"how many ancestor classes can potentially affect this class" - [ChKe91]*. The implications of this metric regard design complexity. If there are deep trees, design complexity increases. This work will measure the absolute position of a class, not the relative distance of a class in the hierarchy against the changed lines value. This is to prove that the deeper a class is in the inheritance hierarchy, the more methods are inherited, thus increasing the complexity of the class.

Numbers of Methods

The Numbers of Methods Metric (NOM) counts the number of methods of a class. There have been several definitions of this metric, [LiHe93] defines it as the number of external methods, while other authors speak of the same NOM, but divide it into two different sets of methods (e.g. number of external methods and number of internal methods vs. number of instance methods and number of class methods). For this study, NOM describes the number of all methods inside a class (this does not include non-overridden inherited methods).

The following hypotheses are being tested against these metrics:

Hypothesis 1.4: A high value of WMC of a class will lead to a high value of changed lines. This is based on the assumption that complexity increases amount of change done.

Hypothesis 1.5: A high value of LCOM* will lead to a high value of changed lines. A low value on LCOM* suggests good design of classes, while a high value hints at too many tasks performed by a class. A bad design of a class should lead to an increase in changed lines.

Hypothesis 1.6: A high value of DIT for a class suggests a class that inherits a lot of methods and therefore is of greater design complexity. A high value of DIT of a class should lead to increased changed lines.

Hypothesis 1.7: A high value of NOM suggests much functionality embedded in a class and therefore high complexity in the class. Opposed to that is the concept of encapsulation where a high number of methods represents low complexity.

Hypothesis 1.7.1: NOM and WMC should have significantly different results as the complexity factor measured in WMC should discard the getter/setter complexity that is skewing NOM.

5 Fitting the Data

5.1 General

To fit the data gathered by the metric tools and the CVS log files, several tools had to be implemented. These tools are written in Java and share the same architecture (a configuration file and a file with the locations of the data to be accessed). Furthermore a script has been implemented to generate a batch file for retrieving all CVS data required. Another tool was implemented that generates Syntax-files for SPSS to help automate the data imports into the statistic software. It operates with an input file and a template file. This step is part of the mining step of the data-mining procedure, except for the script and the SPSS Syntax tool, which are parts of the pre-processing of the data. The goal is to provide the results of the metric suites in a fashion that can be worked on efficiently.

5.2 CVS Parser

The first step to gather the CVS data is to download the projects by using the script introduced in Section 5.5. After the projects have been downloaded, a CVS log command is run on the whole project and saved as “logfile.log”. The directory of the project is added in the file list for the CVS parser. The name for the log file is not configurable at the moment. This is a typical log entry:

```
1. =========================================================
   ================

2. RCS                                                  file:
   /cvsroot/druid/druid/build/src/druid/AntTask.java,v
3. Working file: Druid/build/src/druid/AntTask.java
4. head: 1.2
5. branch:
6. locks: strict
7. access list:
8. symbolic names:
9. V3_8: 1.2.2.2
10.     V3_7: 1.2.2.1
11.     V3_6: 1.2
12.     V3_5: 1.2
13.     Root_V3_X: 1.2
14.     V3_X: 1.2.0.2
15.     V3_4: 1.2
16.     V3_2: 1.2
```

```
keyword substitution: kv
total revisions: 4; selected revisions: 4
description:
----------------------------
revision 1.2
date: 2003/09/02 02:25:15; author: antoniog; state: Exp; lines: +5 -5
branches: 1.2.2;
removing unused and organizing imports
----------------------------
revision 1.1
date: 2003/08/26 17:40:55; author: acarboni; state: Exp;
Added files
----------------------------
revision 1.2.2.2
date: 2006/01/17 01:54:51; author: antoniog; state: Exp; lines: +0 -2
Remove trailing empty lines at the end of the files.
----------------------------
revision 1.2.2.1
date: 2005/12/22 07:05:51; author: antoniog; state: Exp; lines: +3 -3
Cleanup code: Don't create unnecesary new Objects.
```

The parser checks if the RCS file contains the attic directory (indicating if the file has been removed from the project). If not, it saves the working file name for the package structure and the number of lines added, deleted and the sum of them. The metrics plug-in ignores interfaces so the Java files themselves have to be parsed as well and CVS entries for interfaces have to be discarded (note that this is just for the files themselves, inherited interfaces are still included in the metrics; just not the interfaces themselves). This is to make matching of metric results to changed lines values easier. Before interfaces are discarded, the package values are calculated. This is done since interfaces do account for package complexity as they are part of it, while an interface does not account for the complexity of a class that inherits it. This is an arguable point of view as one could point out that a class incorporates the complexity of the interface by implementing it. The implications for design complexity are different from the structural complexity though and for this work the interfaces are part of the design complexity and not part of the structural complexity.

The output of the parser has the following form:

For a class:
```
AntTask.java|8|10|18|Druid.build.src.druid
```

First column is the filename, then the added lines, followed by deleted lines and the sum of added and deleted lines. Last entry is the package name generated from the directory of the file.

For a package:
```
druid|110,0
```

First column is the package name, second column is the sum of the sums of the classes in the package. The package name is gathered from shortening the package name generated from the directory. The output is saved to two files in the format “directory name.txt” for the classes and “directory name + Package.txt” for the packages.

5.3 JDepend Report Parser

JDepend reports are generated by using the JDepend command line and are saved in the format “project name + report”. The JDepend Report Parser loads a file list holding all the generated reports. The reports are in XML format and are of the following form:

```
<Package name="com.lowagie.servlets">
            <Stats>
                <TotalClasses>4</TotalClasses>
                <ConcreteClasses>3</ConcreteClasses>
                <AbstractClasses>1</AbstractClasses>
                <Ca>0</Ca>
                <Ce>12</Ce>
                <A>0.25</A>
                <I>1</I>
                     <D>0.25</D>
                     <V>1</V>
                 </Stats>
    </Package>
```

Output not of interest for the parser has been removed. A DOM parser is used to retrieve the information from the file. The metrics of interest are held in the elements Ca (Afferent Coupling), Ce (Efferent Coupling) and A (Abstractness). The values given are for whole packages. The output is saved into the Reports directory. Each project has its own report in the format “project name + Report.txt”.

The output itself is in the format:

```
gj.awt.geom|V|1|D|0,2|I|0,43|A|0,38|Ce|3|Ca|4|AbstractClass
es|3|ConcreteClasses|5|TotalClasses|8|
```

First part is the package name followed by the nine metric/value pairs analyzed by JDepend.

5.4 Metrics Report Parser

The Metrics Report parser uses a configuration file that holds the metrics of interest for this work. These are WMC, LCOM, DIT and NOM. It also uses a file list to load the reports generated by the metrics plug-in. The metrics output is saved as XML and the Metrics Report Parser uses a DOM parser to go through the reports. The output of the parser has the following form:

```
NOM|Generator.java|Generator|9,0|Druid.build.src.mod.datage
n.generic.torque
```

First column holds the name of the metric, second column holds the file name, followed by the class name and the metric value in column three and four. The last column is the package name generated from the local path of the file. This is of relevance for the matching heuristic used later on to ease analyzing the data.

5.5 Matching Heuristic and Script Generator

The package names constructed from the CVS parser do not always match the proper package; if there is, e.g. a test and a main directory containing the same files in different states, the CVS parser will deliver two files that can only be differentiated by their file names. To be able to assign the right packages to the right CVS data, a heuristic was required that matches the package names to the directory names. The matcher ranks the highest matching filenames by similarity and then the according package is chosen manually.

The script generator creates a batch file that automatically downloads all projects defined in the generators file list. Only the internal SourceForge project name is required, the rest is generated by the generator. The output looks like this:

```
1. md foureverBase
2. cd foureverBase
```

3. `cvs -d:pserver:anonymous@fourever.cvs.sourceforge.net:/cvsroot/fourever checkout .`
4. `cvs log > logfile.log`
5. `cd ..`

There is one entry for every project defined in the file list of the generator.

Another script generator is used to create the syntax for SPSS to automate the processing of the data. It will substitute a place-holder with the SourceForge name of the projects and will generate a list of commands using a template file. Inputs for this are the template to be generated and a list of projects. Output is a syntax file for SPSS that contains the template generated for all projects.

6 Presenting the Results

6.1 General

Out of the 100 projects selected for this study, only 46 could be actually used to provide data. This was unexpected as a sample of 10 projects, finished in the fore field of this study, did not come across the problems encountered in this second, larger study. The issues mainly stemmed from the program structures. The metric suites only deliver relative results, that is, results are presented package-wise (e.g. the class org.main.Run can be found in the directory "project/Test" as well as in "project/Live"; the metric results will not be distinguishable). Many projects however hold different versions of the same project or a test-branch in its path and changes were registered in all of these. The classes share the names and the metric values; however they have completely different values in the Changed Lines Metric. Since it is not predictable with 100% accuracy which package is the original one, double entries had to be eliminated. If projects contained too many of those double packages, the projects were discarded. The threshold for this was chosen partly arbitrarily; when the effort to sort the classes became too large, or more than half of the project was doubled, it was discarded. Furthermore if a project contained more than one version of the project as a branch, the project was discarded as well.

Since the distribution of the CVS data resembled an exponential distribution, the data was logarithmized using a logarithm on the base of 10 and adding a constant value of 1 to every dataset. This yields the following histograms for NCSS, Sum and SumNCSS.

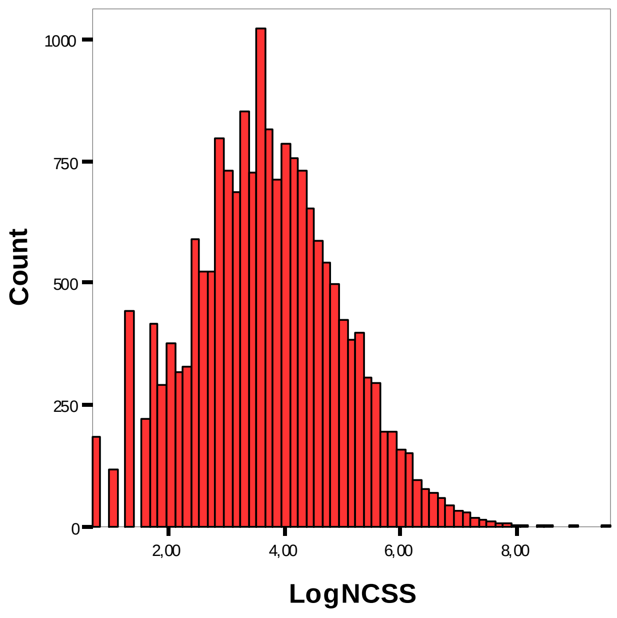

Figure 4: Histogram of NCSS (logarithmized)

Figure 5 shows the problem when using only Sum as dependent variable. A third of the values mined from the CVS are of a zero value; that means no changes were registered in the CVS. This does not seem a problem, as these classes were most likely rather simple classes, added after they were developed locally. However this does pose a problem as they are heavily biasing the results of any regression analysis, requiring a normal distribution of the data.

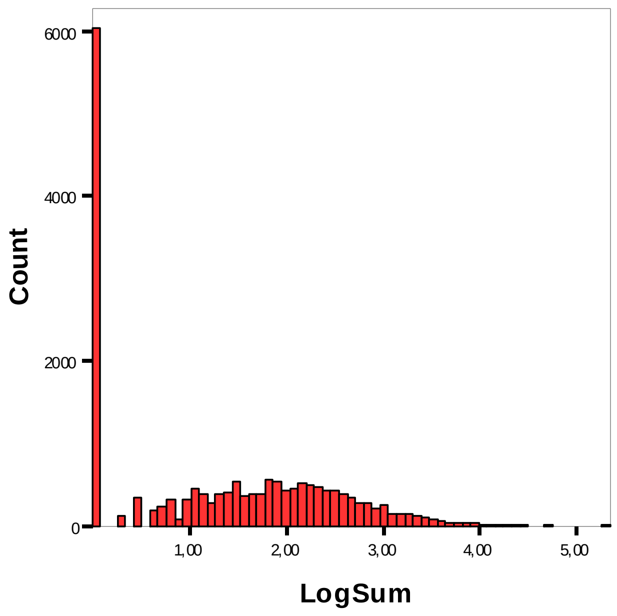

Figure 5: Histogram of Sum (logarithmized)

To be able to analyze all data, NCSS was added to Sum under the assumption that class size accounts as a one-time change (as an add operation to be precise). After undertaking this step the following histogram is gathered.

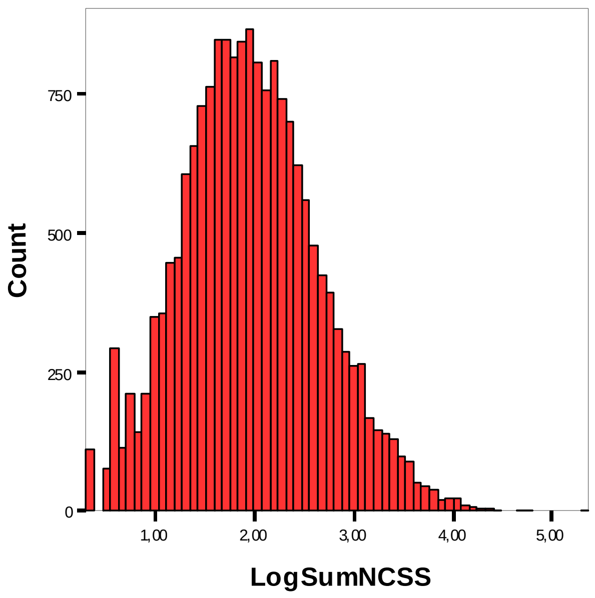

Figure 6: Histogram of SumNCSS (logarithmized)

Based on this histogram it is possible to perform a linear regression analysis. The correlation analysis for Sum and SumNCSS will be done with Pearson's correlation.

The results presented here are created by using SPSS [StPa]. The statistical analysis is done with SPSS as well.

The representation is handled in two parts. The first part shows the correlation of the metric with SumNCSS, Sum and NCSS. Then a regression analysis is done to see if a model can be found that is able to predict values for SumNCSS and Sum.

Kolmogorov-Smirnov Tests for SumNCSS, Sum and NCSS showed significant results, even after being transformed. Significant results with the Kolmogorov-Smirnov Test imply that the data is not normally distributed, however the larger the sample is, the more likely the Kolomogorov-Smirnov Test is to fail [FiAn05]. In previous steps of the study, Spearman's correlation was used instead of Pearson's correlation, however after transforming the data, the results of Pearson's and Spearman's were remarkably close and the histograms for SumNCSS and, to a lesser extent, Sum after being logarithmized actually fitted a normal distribution quite well. So it was possible to use regression analysis on the logarithmized data.

The significance level for this study was chosen at $p < 0,001$ as the sample size is big enough.

6.2 Hypotheses tested

Hypothesis 1.1: A high value of Efferent Coupling of a package will lead to a high value of changed lines. This is based on the assumption that instable packages are changed easier and therefore are changed more often [MaRo03].

Hypothesis 1.2: A high value of Afferent Coupling of a package will lead to a low value of changed lines. According to [MaRo03] a stable package is harder to change and therefore will change less.

Hypothesis 1.3: A high value of Abstractness suggests a stable package (refer to the earlier part of this chapter). Therefore a high value of Abstractness should lead to a low value of changed lines.

Hypothesis 1.3.1: A high value of Abstractness suggests a stable package and should therefore have a high value of Afferent Coupling. Refer to appendix for chart.

Hypothesis 1.4: A high value of WMC of a class will lead to a high value of changed lines. This is based on the assumption that complexity increases amount of change done.

Hypothesis 1.5: A high value of LCOM* will lead to a high value of changed lines. A low value on LCOM* suggests good design of classes, while a high value hints at too many tasks performed by a class. A bad design of a class should lead to an increase in changed lines.

Hypothesis 1.6: A high value of DIT for a class suggests a class that inherits a lot of methods and therefore is of greater design complexity. A high value of DIT of a class should lead to increased changed lines.

Hypothesis 1.7: A high value of NOM suggests much functionality embedded in a class and therefore high complexity in the class. Opposed to that is the concept of encapsulation where a high number of methods implies low complexity.

Hypothesis 1.7.1: NOM and WMC should have significantly different results as the complexity factor measured in WMC should discard the getter/setter complexity that is skewing NOM.

6.3 Class-size

In the first analysis, class-size will be compared to the changes of the class. This is because of the dependent variables, planned to be used in the examination of the metrics.

Therefore the nature of the relationship of Sum and NCSS needs to be analyzed. The first diagram shows Non-Commenting Source Statements measured against the changed lines of the CVS. To be able to use Pearson's correlation and the linear regression analysis, the data was logarithmized.

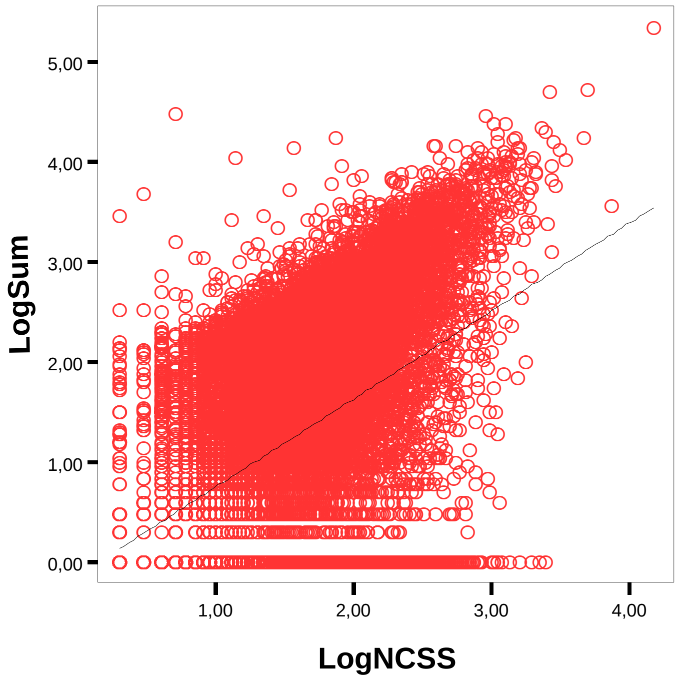

Figure 7: Non-Commenting Source Statements (NCSS) vs. Sum of CVS changes

As can be seen in the diagram, there seems to be a connection between the sums and the class sizes. This was anticipated, as it only makes sense that bigger classes cause more changes.

6.3.1 Correlations

Correlations

		LogNCSS
LogSum	Pearson Correlation	,431**
	Sig. (1-tailed)	,000
	N	18191

**. Correlation is significant at the 0.01 level

Table 3: Correlations of Sum and NCSS

The Correlation Coefficient equals 0,431 and the significance that the two values are related is high (p < 0,001). The value of 0,431 suggests a low positive correlation. This is unexpected as one would naturally assume that the plain size of a class would have more of a visual impact on changes in a class. Even though this is the case, the effect is only minor.

The next analysis was done on package level, as the design metrics will be worked at this level. Here the scatterplot reveals an even more distinct pattern of changes in the CVS measured against the class sizes.

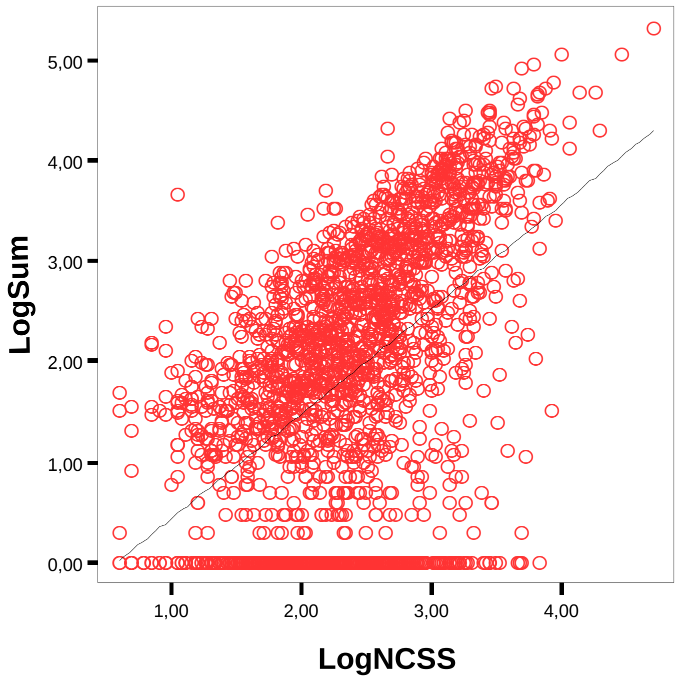

Figure 8: Sum of CVS changes versus Non-Commenting Source Statements on package level

The corresponding correlations table is shown in Table 4:

Correlations

		LogNCSS
LogSum	Pearson Correlation	,523**
	Sig. (1-tailed)	,000
	N	2243

**. Correlation is significant at the 0.01 level

Table 4: Correlations of NCSS and Sum on package level

The correlation analysis on package level yields more favourable results for the assumption of a relationship between size of a unit and changes performed on the unit. The correlation of 0,523 suggests an average positive correlation.

6.3.2 Regression

As it is of interest to further examine how Sum and NCSS relate to each other, a regression analysis was done with Sum as independent variable and NCSS as dependent variable.

Model Summary[b]

Model	R	R Square	Adjusted R Square	Std. Error of the Estimate
1	,431[a]	,186	,186	1,00263

a. Predictors: (Constant), LogNCSS

b. Dependent Variable: LogSum

Table 5: Regression of Sum and NCSS per class

With 0,186 as R Square 18,6% of the variance in NCSS are covered by Sum. This suggests a low linear relationship. The scatterplot of standardized residuals plotted against standardized predicted values carries the assumption of the linear regression model.

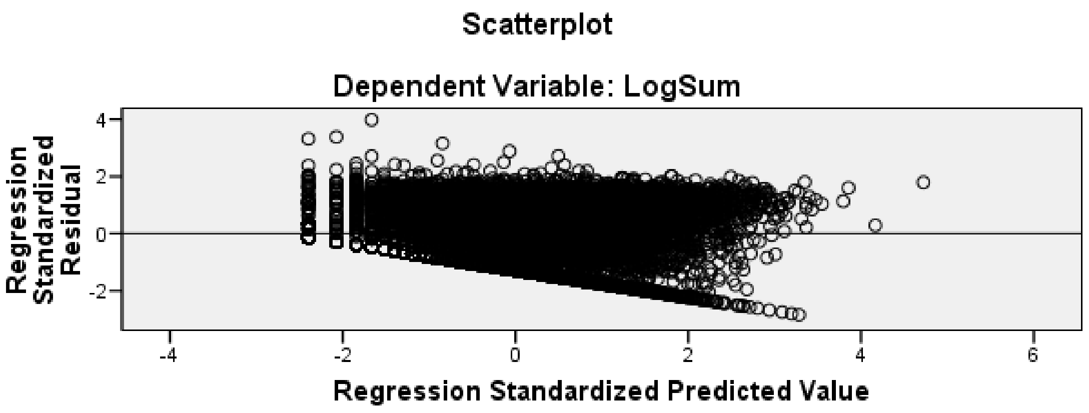

Figure 9: Scatterplot of Residuals and Predicted Values for Sum and NCSS per class

Using linear regression on package level leads to similar results, even though slightly better than on class level.

Model Summary[b]

Model	R	R Square	Adjusted R Square	Std. Error of the Estimate
1	,523[a]	,274	,273	1,12236

a. Predictors: (Constant), LogNCSS

b. Dependent Variable: LogSum

Table 6: Regression of Sum and NCSS per package

NCSS covers about 27% of the variance in Sum and is therefore an average predictor for Sum. The scatterplot of Residuals and Predicted Values confirms this:

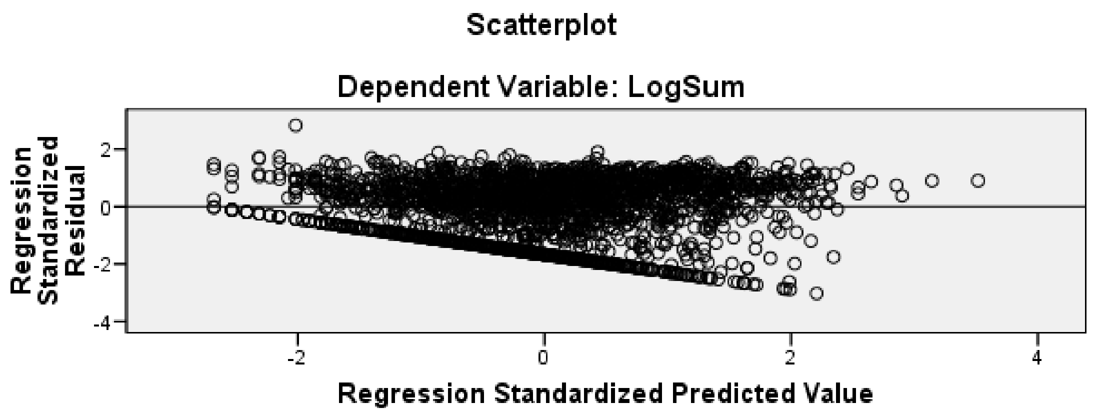

Figure 10: Scatterplot of Residuals and Predicted Values for Sum and NCSS per package

6.4 NOM

The first metric performance examined was that of Number-Of-Methods (NOM). The scatterplot shown in Figure 11 reveals a trend according to Hypothesis 1.7 (A high number of NOM will go with a high number of the SumNCSS value).

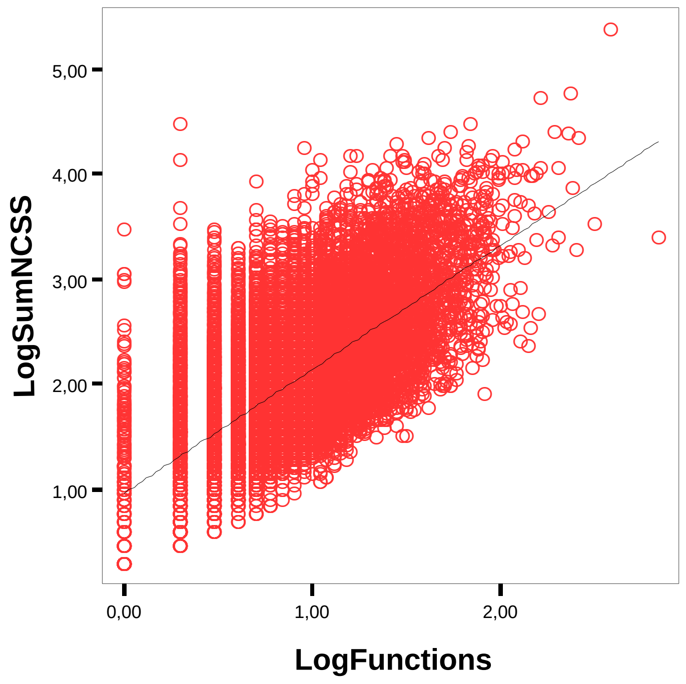

Figure 11: Scatterplot of SumNCSS and NOM (Functions) with regression line

6.4.1 Correlation

Hypothesis 1.7 holds true considering the following correlation of the Sum of Changed Lines and NCSS (further referenced as SumNCSS) with the number of functions per class:

Correlations

		LogFunctions
LogSumNCSS	Pearson Correlation	,641**
	Sig. (1-tailed)	,000
	N	18191

**. Correlation is significant at the 0.01 level

Table 7: Correlations of SumNCSS and Functions (Methods) per Class

The correlation of 0,641 suggests an average positive correlation value. This goes along with Hypothesis 1.7 claiming that on the one hand a high number of functions suggest high complexity encapsulated in a class a high number of functions, while on the other hand also suggests the use of getter/setter methods that implies low complexity. Seeing as the trend is positive and of average value, Hypothesis 1.7 holds true.

The next table illustrates the correlation between the NOM-Metric (Column Functions) and Sum, NCSS and SumNCSS.

Correlations

		LogSum	LogNCSS	LogSum NCSS
LogFunctions	Pearson Correlation	,338**	,785**	,641**
	Sig. (1-tailed)	,000	,000	,000
	N	18191	18191	18191

**. Correlation is significant at the 0.01 level (1-tailed).

Table 8: Correlations of Sum, NCSS, SumNCSS and Functions per Class

Table 8 shows that the strongest correlation of the NOM-Metric is actually with NCSS, while Sum only shows a rather low correlation of 0,338. This is due to a lot of classes having no changes at all and thus heavily skewing the result (the range here is 1 – 2600 NCSS). If the NCSS of a class are treated as a one-time addition effort however, the correlation of the metric is closer to what one would expect.

6.4.2 Regression SumNCSS

To predict values for SumNCSS linear regression was used. The results are:

Model Summary[b]

Model	R	R Square	Adjusted R Square	Std. Error of the Estimate
1	,641[a]	,411	,411	,53067

a. Predictors: (Constant), LogFunctions

b. Dependent Variable: LogSumNCSS

Table 9: Regression of SumNCSS (dependent variable) and NOM (independent variable)

SumNCSS is the dependent variable and NOM is the independent variable. R represents the correlation after Pearson. R Square represents the percentage of the variance in the dependent variable explained by the independent variable in the regression model [LiHe93]. In this case NOM is able to cover 41,1% of the variance in the SumNCSS metric. This value is considered average when trying to make predictions from this model. A trend can be perceived and the value of 0,411 is high enough to make predictions. This suggests that encapsulation is less of a factor than anticipated by Hypothesis 1.7.

An indicator for linear distribution is if the residuals and the predicted value are grouped randomly around the zero line, Figure 12 shows this pattern.

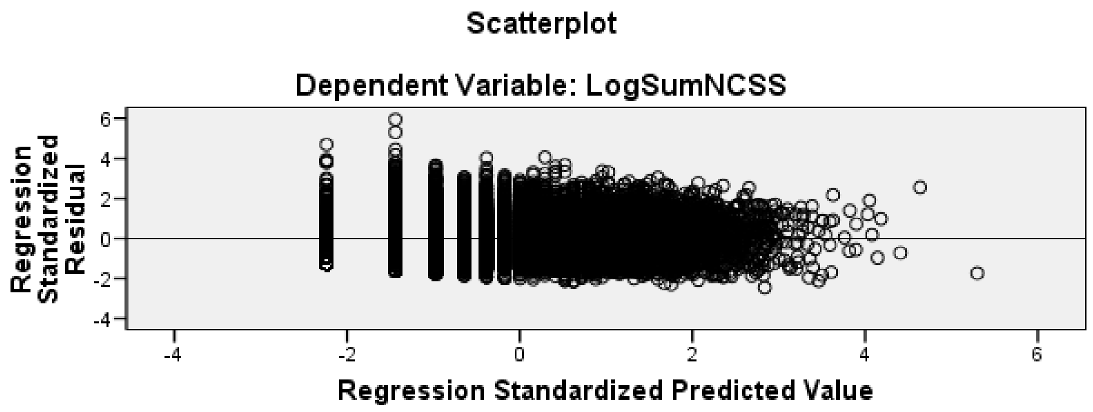

Figure 12: Scatterplot of Predicted Values and Residuals for NOM and SumNCSS

6.4.3 Regression Sum

The results and the scatterplot (see Apendix Figure 28) for the residuals and the predicted values for NOM and Sum are lower than the values in the analysis of SumNCSS.

Model Summary[b]

Model	R	R Square	Adjusted R Square	Std. Error of the Estimate
1	,338[a]	,114	,114	1,04573

a. Predictors: (Constant), LogFunctions

b. Dependent Variable: LogSum

Table 10: Regression of Sum (dependent variable) and NOM (independent variable)

While these values seem more in accordance with Hypothesis 1.7 it must be remarked that the low predictive value stems mostly from the zero values.

6.5 WMC

The next metric of our analysis is the Weighted-Methods-per-Class Metric (WMC). Hypothesis 1.4 states that a high value of WMC will lead to a high value of changed lines (SumNCSS). The scatterplot for WMC and SumNCSS is shown in Figure 8:

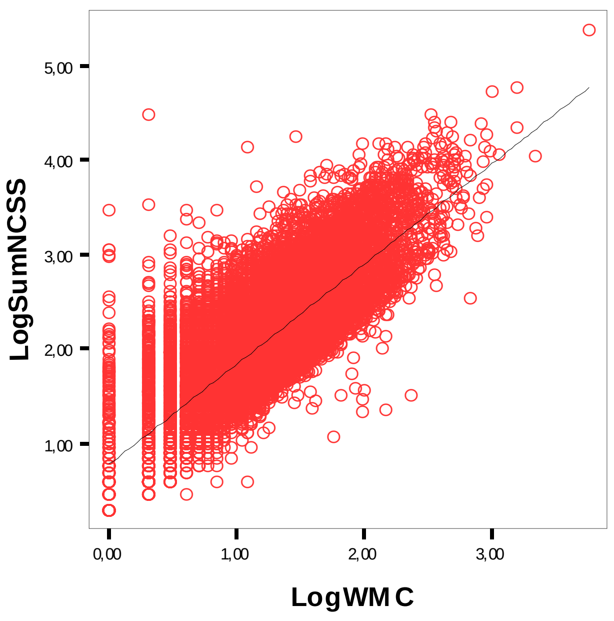

Figure 13: WMC and SumNCSS per Class with regression line

The scatterplot looks similar to the one presented in Figure 12, however a little bit more steep, suggesting an even better correlation than the NOM-Metric. This is in accordance with Hypothesis 1.7.1 that states that WMC should be a better indicator for change than NOM due to the use of getter and setter methods in Java.

6.5.1 Correlation

Table 11 presents the correlation of WMC and SumNCSS:

Correlations

		LogWMC
LogSumNCSS	Pearson Correlation	,773**
	Sig. (1-tailed)	,000
	N	18191

**. Correlation is significant at the 0.01 level

Table 11: Correlation of WMC (Value) and SumNCSS

The correlation value of 0,773 suggests a strong correlation of WMC and SumNCSS and therefore Hypothesis 1.4 and Hypothesis 1.7.1 hold true. The next table shows correlations of WMC with Sum, NCSS and SumNCSS.

Correlations

		LogSum	LogNCSS	LogSum NCSS
LogWMC	Pearson Correlation	,420**	,936**	,773**
	Sig. (1-tailed)	,000	,000	,000
	N	18191	18191	18191

**. Correlation is significant at the 0.01 level (1-tailed).

Table 12: Correlations WMC, Sum, NCSS and SumNCSS

The correlation of the metric with NCSS is actually higher than the one with SumNCSS. However both correlations (NCSS and Sum) are higher than with the NOM metric, suggesting that WMC fits the change effort better than NOM (in all cases). Of interest as well is the correlation value of 0,936 of WMC and NCSS. While there is a very strong correlation of NCSS and WMC, this is not the focus of this work as it is not a measure of maintenance effort, but just a curious detail worth pointing out.

6.5.2 Regression SumNCSS

If Hypotheses 1.7.1 holds true, WMC should give a better prediction (that is a higher percentage of variance in the dependent variable explained by the independent variable) than NOM. As can be seen in Table 13, this holds true:

Model Summary[b]

Model	R	R Square	Adjusted R Square	Std. Error of the Estimate
1	,773[a]	,598	,598	,43818

a. Predictors: (Constant), LogWMC

b. Dependent Variable: LogSumNCSS

Table 13: Regression of SumNCSS (dependent variable) and WMC (independent variable)

WMC delivers a higher R Square value and can actually cover 59,8% of the variance perceived in the dependent variable (SumNCSS).

Since WMC shows a strong positive correlation with SumNCSS, Hypothesis 1.4 holds true. Furthermore linear regression seems a good model for predicting values of SumNCSS with WMC. Plotting the predicted values with the residuals shows that the relationship is linear (the residuals are randomly distributed around the zero line).

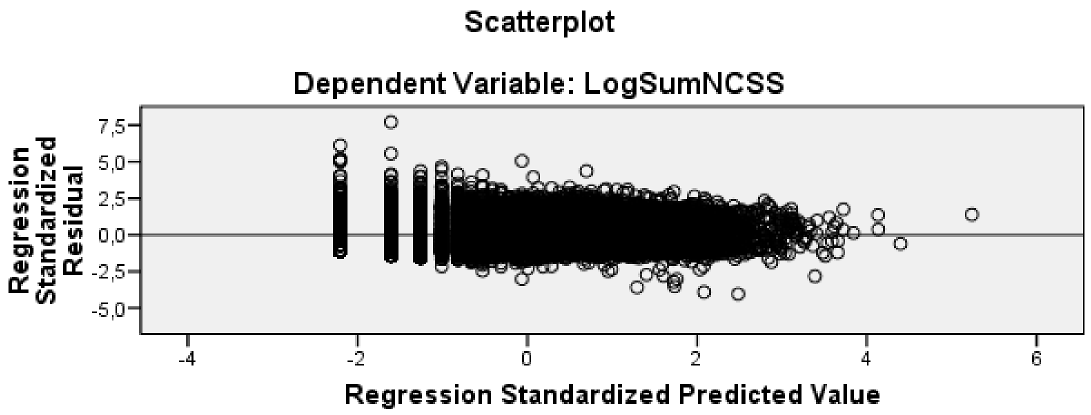

Figure 14: Scatterplot of Predicted Values and Residuals of WMC and SumNCSS

6.5.3 Regression Sum

The regression analysis performed with Sum as the dependent variable reveals a similar trend than with SumNCSS. The predictive power of WMC using a linear regression model is however lower than with SumNCSS (0,176 as opposed to 0,598), again the implication is that the zero values skew the model.

Model Summary[b]

Model	R	R Square	Adjusted R Square	Std. Error of the Estimate
1	,420[a]	,176	,176	1,00852

a. Predictors: (Constant), LogWMC

b. Dependent Variable: LogSum

Table 14: Regression of Sum (dependent variable) and WMC (independent variable)

Bearing the zero values in mind, the same conclusion for Hypothesis 1.4 can be drawn from this regression analysis.

6.6 LCOM*

After showing the results of WMC and NOM, the last code metric is presented. From the code metrics, this metric was expected to deliver the least significant results of the three as it is not a good metric to measure object-oriented design. This has been discussed in Chapter 4. This work used the added LCOM* values of a file in case of inner classes as Sum is only available for a whole file. Figure 15 shows the scatterplot of LCOM* and SumNCSS.

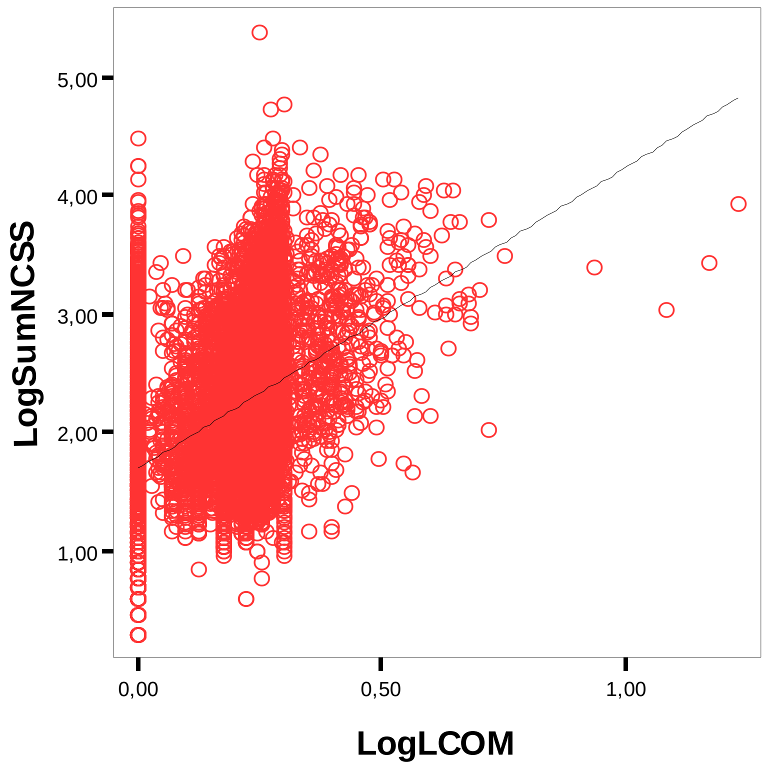

Figure 15: LCOM and SumNCSS per Class and regression line

A positive trend as with the other metrics can be perceived, however the bandwidth of zero LCOM* values has a rather big range (1 – 30.000 SumNCSS).

6.6.1 Correlations

This should be represented in the correlations calculated in Table 15:

Correlations

		LogSum NCSS
LogLCOM	Pearson Correlation	,464**
	Sig. (1-tailed)	,000
	N	18188

**. Correlation is significant at the 0.01 level

Table 15: Correlation of LCOM* and SumNCSS

LCOM* delivers the lowest correlations of the code metrics with SumNCSS. The correlation of 0,464 is considered a low correlation. Significance is high. Table 16 shows the correlations of Sum, NCSS and SumNCSS.

Correlations

		LogSum	LogNCSS	LogSum NCSS
LogLCOM	Pearson Correlation	,259**	,572**	,464**
	Sig. (1-tailed)	,000	,000	,000
	N	18188	18188	18188

**. Correlation is significant at the 0.01 level (1-tailed).

Table 16: Correlations LCOM (Value) with Sum, NCSS and SumNCSS

As with the first two metrics, LCOM* too has a lower correlation with Sum than with the NCSS. The correlation of LCOM* and NCSS is of an average level, while Sum is considered a low correlation. All correlations are significant at $p < 0,001$.

6.6.2 Regression SumNCSS

As has been expected LCOM shows a low predictive capability. A linear regression model is used.

Model Summary[b]

Model	R	R Square	Adjusted R Square	Std. Error of the Estimate
1	,464[a]	,215	,215	,61241

a. Predictors: (Constant), LogLCOM

b. Dependent Variable: LogSumNCSS

Table 17: Regression of SumNCSS (dependent variable) and LCOM (independent variable)

The R Square value of 0,215 makes LCOM* an average predictor for SumNCSS, performing worse than NOM and WMC. Plotting the standardized residuals against the standardized predicted values shows that the zero values of LCOM* skew what otherwise might be a randomly distributed sample. This further indicates why LCOM* scored lower than WMC and NOM.

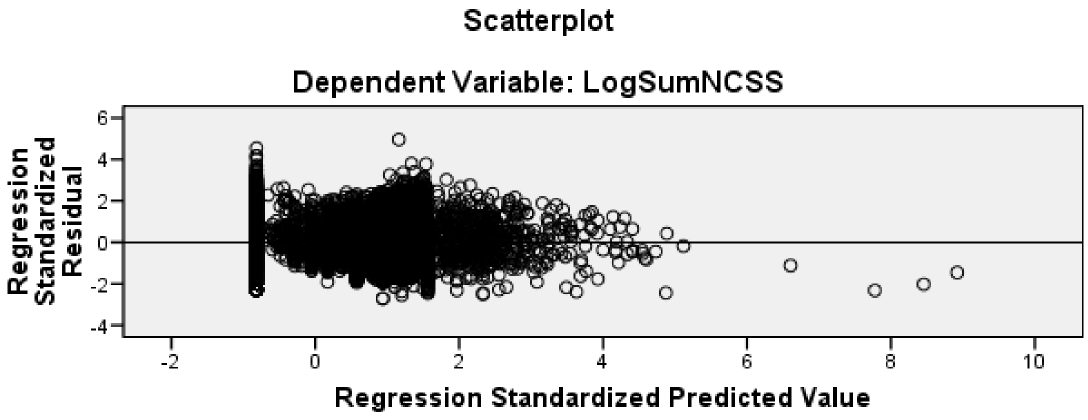

Figure 16: Scatterplot of residuals and predicted values

Hypothesis 1.5 holds true for SumNCSS. An average positive correlation suggests that a high value of LCOM* will likely go with a high value of SumNCSS. Due to LCOM* covering 21,5% of the variance in SumNCSS, it is an average predictor for SumNCSS.

6.6.3 Regression Sum

Using linear regression to predict Sum values for LCOM* does not net any viable results. R Square is of a much lower value than with SumNCSS and at 0,067 only 6,7% of the variance in Sum are accounted for by LCOM*. The scatterplot for the standardized residuals and standardized predicted values is in the Appendix (Figure 30).

Model Summary[b]

Model	R	R Square	Adjusted R Square	Std. Error of the Estimate
1	,259[a]	,067	,067	1,07290

a. Predictors: (Constant), LogLCOM

b. Dependent Variable: LogSum

Table 18: Regression of Sum (dependent variable) and LCOM (independent variable)

6.7 CE/CA

After presenting the results for the code metrics, Efferent and Afferent Coupling (CE and CA) of the structure metrics are the next to be examined.

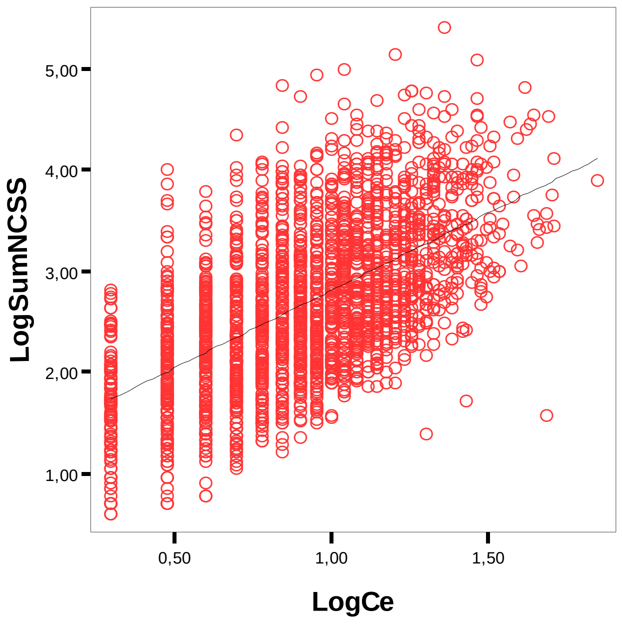

Figure 17: CE and SumNCSS per package with regression line

The scatterplot shows a recognizable trend of a high value of CE coupled with a high value of SumNCSS. This is reflected by the calculation of the correlation as seen in Table 19. Before the correlations are presented the scatterplot of SumNCSS and CA is shown in Figure 18:

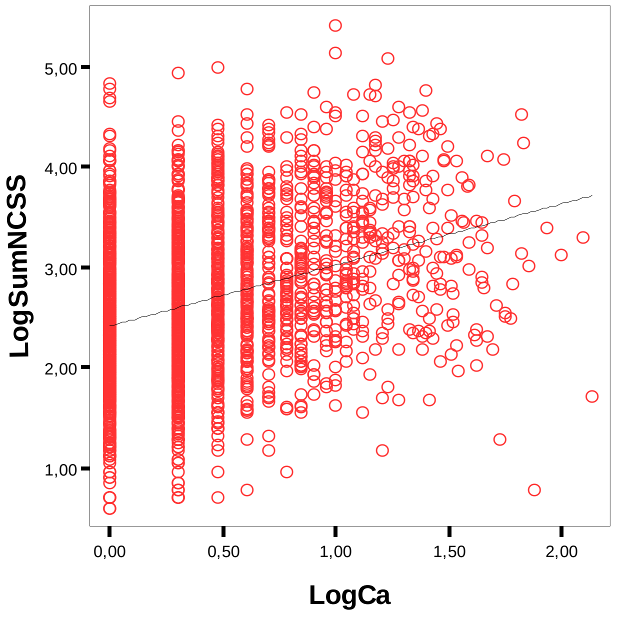

Figure 18: CA and SumNCSS per package with regression line

A trend as with Efferent Coupling is not recognizable. The scatterplot is randomly distributed and the regression line is not steep.

6.7.1 Correlations

Table 19 shows the correlations for CE and CA with SumNCSS:

Correlations

		LogSum NCSS
LogCe	Pearson Correlation	,585**
	Sig. (1-tailed)	,000
	N	2243
LogCa	Pearson Correlation	,336**
	Sig. (1-tailed)	,000
	N	2243

**. Correlation is significant at the 0.01 level

Table 19: Correlations of CE and CA with SumNCSS

As before the results of the correlations for Sum, NCSS and SumNCSS are presented as well:

Correlations

		LogSum	LogNCSS	LogSum NCSS
LogCe	Pearson Correlation	,370**	,653**	,585**
	Sig. (1-tailed)	,000	,000	,000
	N	2243	2243	2243
LogCa	Pearson Correlation	,278**	,312**	,336**
	Sig. (1-tailed)	,000	,000	,000
	N	2243	2243	2243

**. Correlation is significant at the 0.01 level (1-tailed).

Table 20: Correlations of CE and CA with Sum, NCSS and SumNCSS

Efferent Coupling shows a pattern similar to the other metrics, that is a low correlation with Sum, a higher correlation with NCSS (although still of an average level) and an average correlation with SumNCSS. Afferent Coupling on the other hand has its highest correlation with SumNCSS, while having a lower correlation through Sum, NCSS and SumNCSS.

6.7.2 Regression SumNCSS

Efferent Coupling (CE) has an average predictive power of 0,342 (34,2% of the variance in SumNCSS accounted for by CE).

Model Summary[b]

Model	R	R Square	Adjusted R Square	Std. Error of the Estimate
1	,585[a]	,342	,342	,63178

a. Predictors: (Constant), LogCe

b. Dependent Variable: LogSumNCSS

Table 21: Regression of CE and SumNCSS

Afferent Coupling (CA) has a lower correlation with SumNCSS than Efferent Coupling. With a low predictive value of 0,113, linear regression might not be the right model to predict SumNCSS by CA. Only 11,3% of the variance in SumNCSS are covered by CA.

Model Summary[b]

Model	R	R Square	Adjusted R Square	Std. Error of the Estimate
1	,336[a]	,113	,113	,73340

a. Predictors: (Constant), LogCa

b. Dependent Variable: LogSumNCSS

Table 22: Regression of CA and SumNCSS

Hypothesis 1.1 holds true. The correlation is of an average threshold and while the predictive quality of the linear regression model is only average as well, there definitely is a positive connection between Efferent Coupling and a high value of SumNCSS. Hypothesis 1.2 has to be discarded. There should be a negative correlation between CA and SumNCSS. However the correlation is low positive and significant. There is no negative trend as expected in Hypothesis 1.2. The R Square value indicates that the predictive power of the linear regression model for CA is low. Plotting the residuals and the predicted values further supports the decline of the linear regression model for CA.

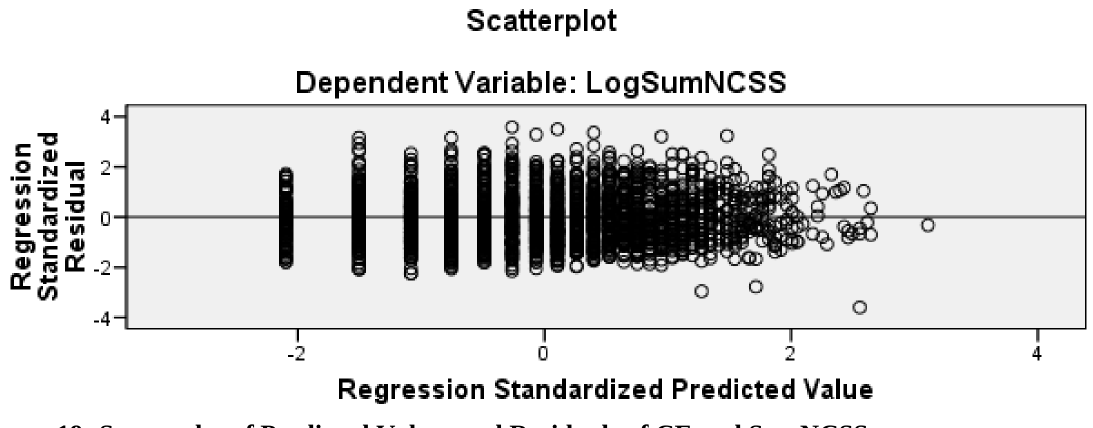

Figure 19: Scatterplot of Predicted Values and Residuals of CE and SumNCSS

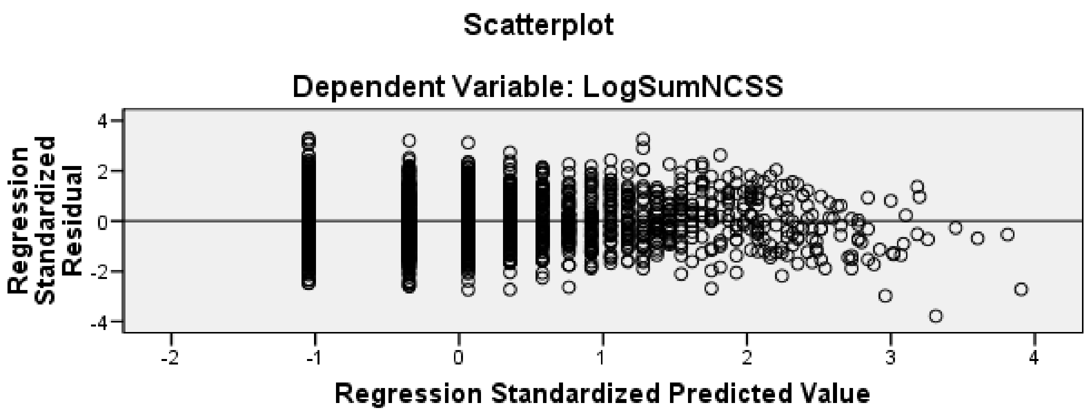

Figure 20: Scatterplot of Predicted Values and Residuals of CA and SumNCSS

The first half of the scatterplots look similar, however the CA scatter plot shows a trend that is leaning toward the right lower half of the scatter plot, while CE revolves around the middle with a few outliers.

6.7.3 Regression Sum

The regression for Sum and Efferent Coupling is following the same pattern as the earlier analyses by performing poorly compared to SumNCSS. A low R Square value shows that a linear regression model is not a good way to predict Sum by Efferent Coupling.

Model Summary[b]

Model	R	R Square	Adjusted R Square	Std. Error of the Estimate
1	,370[a]	,137	,137	1,22320

a. Predictors: (Constant), LogCe

b. Dependent Variable: LogSum

Table 23: Regression of Sum (dependent variable) and Efferent Coupling (independent variable)

Afferent Coupling is much closer to Efferent Coupling in predicting Sum. However it still shows a positive trend where there should be a negative, so Hypothesis 1.2 has to be discarded, while Hypothesis 1.1 holds true as there is a significant, if low, positive trend. However CE cannot be used as a predictor for Sum. Again linear regression is not the means to predict any values for Sum by Afferent Coupling. The according scatterplots of standardized residuals and standardized predictions can be found in the Appendix (Figure 31 and Figure 32).

Model Summary[b]

Model	R	R Square	Adjusted R Square	Std. Error of the Estimate
1	,278[a]	,077	,077	1,26482

a. Predictors: (Constant), LogCa

b. Dependent Variable: LogSum

Table 24: Regression of Sum (dependent variable) and Afferent Coupling (independent variable)

6.8 Depth of Inheritance Tree

The next metric is Depth of Inheritance Tree.Figure 21 shows the scatterplot for SumNCSS and the DIT values delivered by the metric.

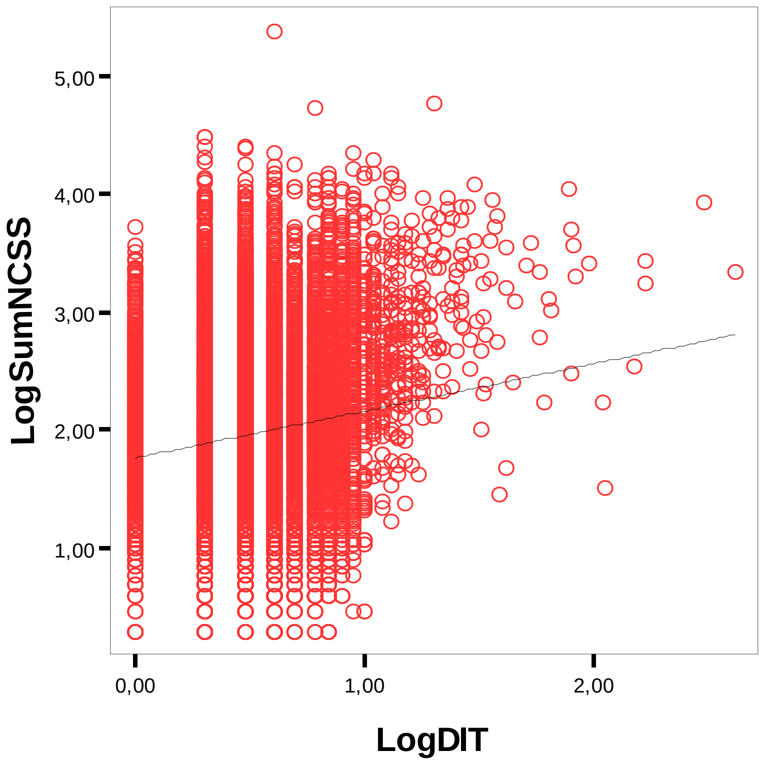

Figure 21: of SumNCSS and DIT

Depth of Inheritance was examined on class level as opposed to Efferent and Afferent Coupling and Abstractness. The scatterplot does not reveal a trend. This is further emphasized by the flat regression line.

6.8.1 Correlations

As with Afferent Coupling a correlation of SumNCSS and DIT is unlikely. The correlations are presented in Table 25:

Correlations

		LogSum NCSS
LogDIT	Pearson Correlation	,149**
	Sig. (1-tailed)	,000
	N	18190

**. Correlation is significant at the 0.01 level

Table 25: Correlation of DIT and SumNCSS

The correlation is highly significant (p < 0,001), but of a very low value. A high value of DIT has only a very low impact on the SumNCSS value and vice versa. As with Afferent Coupling, the correlations of DIT with Sum, NCSS and SumNCSS show a different trend than the previous results:

Correlations

		LogSum	LogNCSS	LogSum NCSS
LogDIT	Pearson Correlation	,115**	,147**	,149**
	Sig. (1-tailed)	,000	,000	,000
	N	18190	18190	18190

**. Correlation is significant at the 0.01 level (1-tailed).

Table 26: Correlations of DIT and Sum, NCSS and SumNCSS

Opposed to the trend of NCSS having a higher correlation than SumNCSS, SumNCSS performs slightly better this time. However it is still a very low value.

6.8.2 Regression SumNCSS

The scatterplot suggests no linear relationship of the data; therefore the expected regression results are of a very low value.

Model Summary[b]

Model	R	R Square	Adjusted R Square	Std. Error of the Estimate
1	,149[a]	,022	,022	,68347

a. Predictors: (Constant), LogDIT

b. Dependent Variable: LogSumNCSS

Table 27: Regression of DIT and SumNCSS

As expected from the scatterplot, linear regression fits the data very badly. With 2% of the variance explained by the model, it is insufficient. After seeing the low correlation this was expected. Plotting residuals and predicted values shows the data to be above the zero line for the most part.

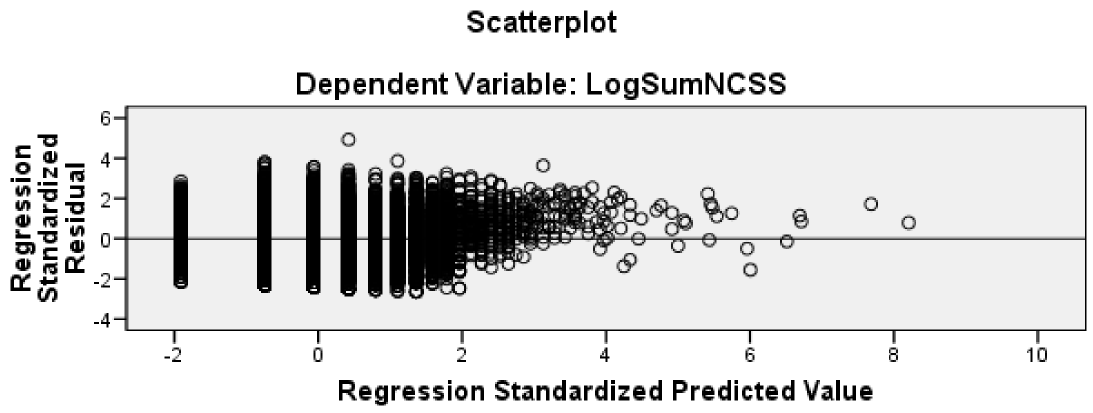

Figure 22: Residuals and Predicted Values of DIT and SumNCSS

Since correlations are very low, Hypothesis 1.6 does not hold true. Furthermore linear regression does not fit the scatterplot. DIT does not go with a high value of changed lines according to the data gathered for this study.

6.8.3 Regression Sum

The results of the linear regression of Sum and DIT are close to the results of SumNCSS. This trend can be perceived in all examinations so far. The same conclusions can be drawn for Sum; linear regression is a bad model to predict Sum by DIT. About 1% of the variance in Sum is covered by DIT. The scatterplot of the residuals and predicted values can be found in the Appendix (Figure 33).

Model Summary[b]

Model	R	R Square	Adjusted R Square	Std. Error of the Estimate
1	,115[a]	,013	,013	1,10365

a. Predictors: (Constant), LogDIT

b. Dependent Variable: LogSum

Table 28: Regression of Sum (dependent value) and DIT (independent value)

6.9 Abstractness

The last metric under examination is the Abstractness metric. The following scatterplot is presented:

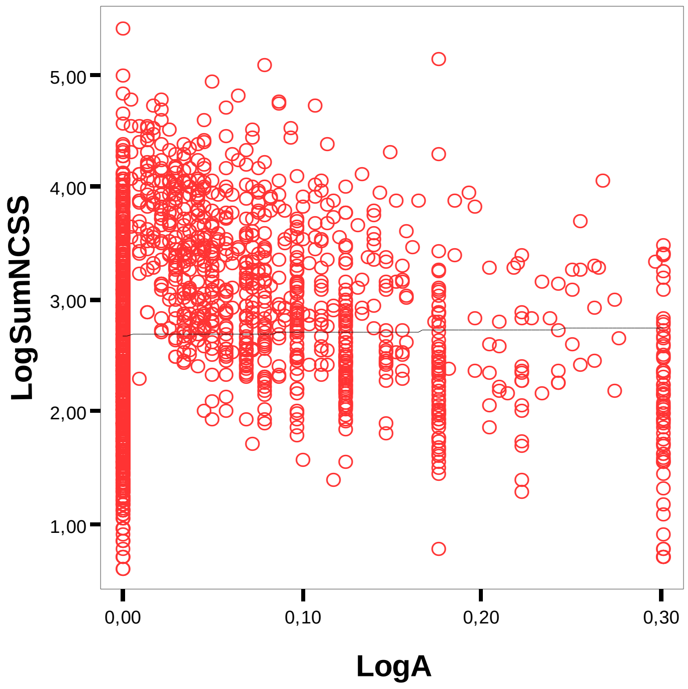

Figure 23: Abstractness (A) and SumNCSS per Class with regression line

The scatterplot reveals that if the zero values were to be removed, the correlation would most likely be negative. As it is, there is no perceivable trend learned from the scatter-plot.

6.9.1 Correlations

Thus no high correlations are expected and none are seen in Table 29 and Table 30.

Correlations

		LogSum NCSS
LogA	Pearson Correlation	,021
	Sig. (1-tailed)	,163
	N	2243

Table 29: Correlation of Abstractness (A) and SumNCSS

This metric was performed on package level again. The correlation is of a low threshold andnot significant at p < 0,001. The correlation of NCSS suggests what could be glanced from the scatterplot; if there were any trend in Abstractness, it would be a nega-tive one. Unfortunately there simply are too many zero values to be able to perform a proper analysis of the data. It is not possible to make any statement about SumNCSS, Sum or NCSS from the data acquired in this study by using Abstractness.

Correlations

		LogSum	LogNCSS	LogSum NCSS
LogA	Pearson Correlation	,052**	-,048*	,021
	Sig. (1-tailed)	,007	,011	,163
	N	2243	2243	2243

**. Correlation is significant at the 0.01 level (1-tailed).

*. Correlation is significant at the 0.05 level (1-tailed).

Table 30: Correlations of Abstractness (A) and Sum, NCSS and SumNCSS

The correlations are not significant at the p < 0,001 level selected for this study.

6.9.2 Regression SumNCSS

With the low correlations it was anticipated that linear regression would not be able to fit the data. Table 31 shows this holds true.

Model Summary[b]

Model	R	R Square	Adjusted R Square	Std. Error of the Estimate
1	,021[a]	,000	,000	,77861

a. Predictors: (Constant), LogA

b. Dependent Variable: LogSumNCSS

Table 31: Regression of Abstractness and SumNCSS

Abstractness explains none of the variance in SumNCSS, since the correlations are not significant at the threshold of p < 0,001 chosen for this study.

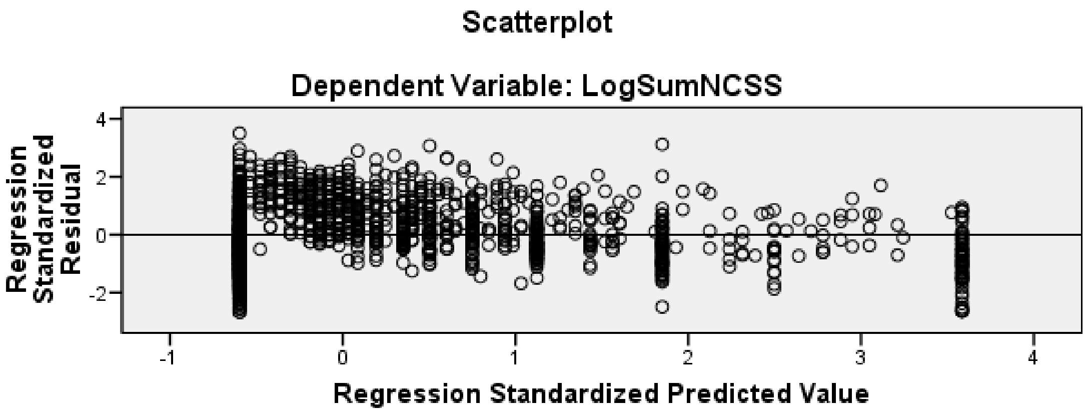

Figure 24: Scatterplot of Residuals and Predicted Values of Abstractness and SumNCSS

Hypthoesis 1.3 is discarded according to the available data, as Abstractness fails to deliver a negative correlation (or any significant correlation).

6.9.3 Regression Sum

Linear regression is unfit to predict any values of Sum by Abstractness. The same conclusions as in the SumNCSS Regression apply here. The scatterplot of the residuals and the predicted values can be found in the Appendix (Figure 34).

Model Summary[b]

Model	R	R Square	Adjusted R Square	Std. Error of the Estimate
1	,052[a]	,003	,002	1,31506

a. Predictors: (Constant), LogA

b. Dependent Variable: LogSum

Table 32: Regression of Abstractness and Sum

To examine Hypothesis 1.3.1 the same analysis is done and regression has been used. Abstractness is the independent variable, while Afferent Coupling is the dependent variable for the regression analysis. The scatterplot with the regression line can be found in the appendix together with the scatterplot of the residuals and the predicted values (Figure 35 and Figure 36). Correlation and linear regression tables can be found in the appendix as well (Table 42 and Table 43). The relationship of A and CA is positive as is suggested by Hypothesis 1.3.1. However the correlation is only of a low value, namely 0,301. This value is sufficient to confirm Hypothesis 1.3.1, as it is a significant, positive correlation. However linear regression is not fit to predict values as R Square is 0,091. This is too low of a value to hold on to a linear model.

6.10 Stepwise multiple linear Regression

6.10.1 Code Metrics

The next step of the analysis was to test for Hypothesis 2; if Hypothesis 1 holds true, is it possible to find a set of methods that is valid for all projects. To examine this, multiple linear regression was used on the metrics per class and on the metrics per package (in two separate runs).

SumNCSS

First the metrics per class were analysed, using SumNCSS as dependent variable.

Model Summary

Model	R	R Square	Adjusted R Square	Std. Error of the Estimate
1	,773[a]	,598	,598	,43816
2	,774[b]	,600	,599	,43735
3	,775[c]	,601	,601	,43645
4	,776[d]	,603	,603	,43568

a. Predictors: (Constant), LogWMC

b. Predictors: (Constant), LogWMC, LogFunctions

c. Predictors: (Constant), LogWMC, LogFunctions, LogLCOM

d. Predictors: (Constant), LogWMC, LogFunctions, LogLCOM, LogDIT

Table 33: Multiple linear Regression SumNCSS (dependent variable) and WMC, NOM (Functions), LCOM and DIT (independent variables)

As can be seen in Table 33, WMC accounts for most of the variance covered by the model. This is reflected by the t-values in Table 34, where WMC has the highest value, while NOM, LCOM and DIT are all low. However all values are highly significant ($p < 0,001$).

The scatterplot of the standardized residuals and standardized predicted values resembles the distribution of WMC and further carries the assumption that WMC makes up for most of the predictive value in the model. Therefore Hypothesis 2 does seem to hold, however WMC alone is nearly just as good as the whole model for SumNCSS. Since the other independent variables are highly significant at $p < 0,001$, they have to be included in the model.

Coefficients[a]

Model		Unstandardized Coefficients B	Unstandardized Coefficients Std. Error	Standardized Coefficients Beta	t	Sig.
1	(Constant)	,784	,008		99,647	,000
	LogWMC	1,060	,006	,773	164,483	,000
2	(Constant)	,802	,008		98,322	,000
	LogWMC	1,147	,012	,837	92,661	,000
	LogFunctions	-,137	,017	-,075	-8,258	,000
3	(Constant)	,822	,008		97,367	,000
	LogWMC	1,128	,013	,823	89,847	,000
	LogFunctions	-,168	,017	-,092	-9,928	,000
	LogLCOM	,276	,032	,051	8,704	,000
4	(Constant)	,774	,010		75,260	,000
	LogWMC	1,116	,013	,814	88,406	,000
	LogFunctions	-,151	,017	-,082	-8,888	,000
	LogLCOM	,237	,032	,043	7,393	,000
	LogDIT	,104	,013	,039	8,116	,000

a. Dependent Variable: LogSumNCSS

Table 34: Multiple linear Regression Coefficients SumNCSS (dependent variables) and WMC, NOM (Functions), LCOM and DIT (independent variables)

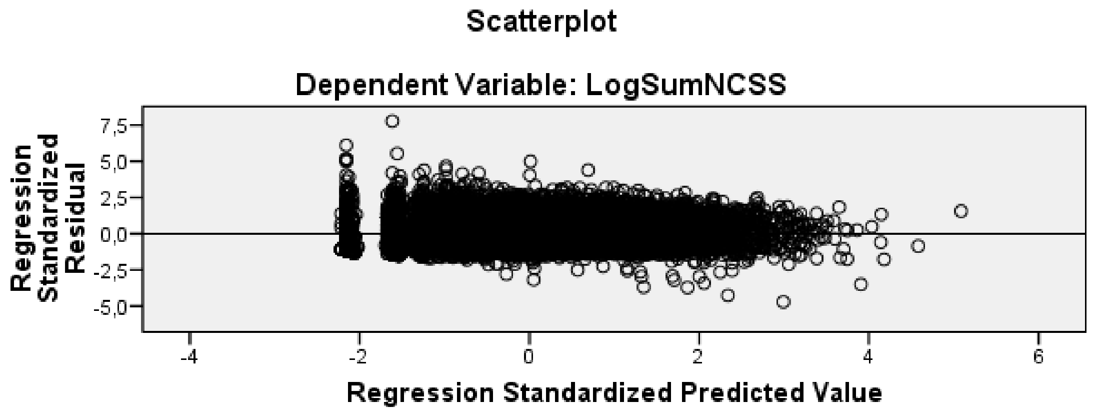

Figure 25: Scatterplot of Residuals and Predicted Values of WMC, NOM, LCOM and DIT and SumNCSS

Hypothesis 2 does hold true, there is a set of metrics viable for all projects, however that set could be replaced by just using the WMC metric. Still the additional metrics make for a higher accuracy of the predictive model.

Sum

Sum shows the same pattern as SumNCSS, with lower predictive quality of the metrics. WMC is covering most of the variance in Sum, while NOM, LCOM and DIT contribute small amounts.

Model Summary[e]

Model	R	R Square	Adjusted R Square	Std. Error of the Estimate
1	,419[a]	,176	,176	1,00852
2	,421[b]	,177	,177	1,00758
3	,423[c]	,179	,179	1,00681
4	,426[d]	,181	,181	1,00520

a. Predictors: (Constant), LogWMC

b. Predictors: (Constant), LogWMC, LogFunctions

c. Predictors: (Constant), LogWMC, LogFunctions, LogLCOM

d. Predictors: (Constant), LogWMC, LogFunctions, LogLCOM, LogDIT

e. Dependent Variable: LogSum

Table 35: Multiple linear Regression Sum (dependent variables) and WMC, NOM (Functions), LCOM and DIT (independent variables)

This holds true when looking at the t-values in Table 36. As before all metrics are of high significance.

Coefficients[a]

Model		Unstandardized Coefficients		Standardized Coefficients	t	Sig.
		B	Std. Error	Beta		
1	(Constant)	,258	,018		14,247	,000
	LogWMC	,924	,015	,419	62,292	,000
2	(Constant)	,288	,019		15,334	,000
	LogWMC	1,068	,029	,485	37,453	,000
	LogFunctions	-,226	,038	-,077	-5,925	,000
3	(Constant)	,316	,019		16,217	,000
	LogWMC	1,041	,029	,472	35,939	,000
	LogFunctions	-,270	,039	-,092	-6,932	,000
	LogLCOM	,394	,073	,045	5,375	,000
4	(Constant)	,211	,024		8,898	,000
	LogWMC	1,014	,029	,460	34,828	,000
	LogFunctions	-,234	,039	-,079	-5,959	,000
	LogLCOM	,308	,074	,035	4,160	,000
	LogDIT	,227	,030	,053	7,691	,000

a. Dependent Variable: LogSum

Table 36: Multiple linear Regression Sum Coefficients (dependent variables) and WMC, NOM (Functions), LCOM and DIT (independent variables)

The scatterplot of the standardized residuals and predicted values shows the WMC pattern and as before, the zero values are skewing the plot.

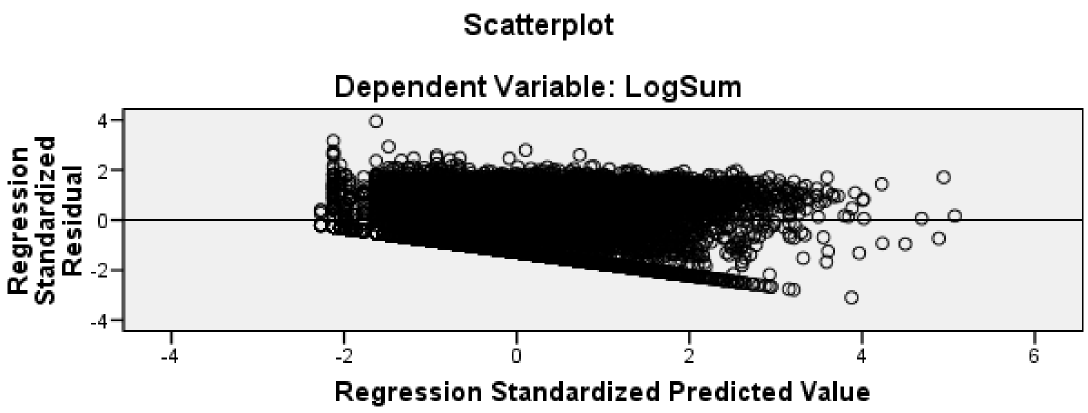

Figure 26: Scatterplot of Residuals and Predicted Values of WMC, NOM, LCOM and DIT and Sum

Hypothesis 2 has to be discarded for Sum as the variance explained by the metrics is too low to justify using the model as predictor for future Sum values.

6.10.2 Structure Metrics

After running the regression analysis on the metrics per class, the next step is to examine the metrics per package; that is Efferent and Afferent Coupling and Abstractness. Hypothesis 2 was tested against SumNCSS and Sum.

SumNCSS

Table 37 shows that CE and CA together provide an average predictor for SumNCSS. Abstractness does not improve the model whatsoever. When looking at the coefficients it becomes obvious that Abstractness is yet again not significant. As with the structure metrics, one metric seems to explain the biggest part of the variance, namely CE.

Model Summary[d]

Model	R	R Square	Adjusted R Square	Std. Error of the Estimate
1	,585[a]	,342	,342	,63178
2	,618[b]	,382	,381	,61233
3	,618[c]	,382	,381	,61243

a. Predictors: (Constant), LogCe

b. Predictors: (Constant), LogCe, LogCa

c. Predictors: (Constant), LogCe, LogCa, LogA

d. Dependent Variable: LogSumNCSS

Table 37: Multiple linear Regression SumNCSS (dependent variable) and CE, CA and Abstractness (independent variables)

The t-values further strengthen this assumption. Furthermore it is shown that Abstractness is not significant; it does not affect the model at all:

Coefficients[a]

Model		Unstandardized Coefficients B	Unstandardized Coefficients Std. Error	Standardized Coefficients Beta	t	Sig.
1	(Constant)	1,270	,044		29,129	,000
	LogCe	1,535	,045	,585	34,120	,000
2	(Constant)	1,223	,042		28,824	,000
	LogCe	1,403	,045	,535	31,221	,000
	LogCa	,374	,031	,207	12,067	,000
3	(Constant)	1,218	,044		27,792	,000
	LogCe	1,408	,046	,536	30,803	,000
	LogCa	,368	,033	,203	11,190	,000
	LogA	,101	,192	,009	,527	,598

a. Dependent Variable: LogSumNCSS

Table 38: Coefficients of multiple linear regression for SumNCSS

CE and CA are highly significant, while according to the t-values most of the predictive quality of the model stems from CE. That is the reason for the low increase in predictive quality. Hypothesis 2 only holds true if CE and CA are treated as two separate metrics, because Abstractness has to be discarded; it does not bring any new explanation of the variance into the model. Furthermore the overall value of R Square is average, suggesting that linear regression fits the data. The explained variance in SumNCSS by CE and CA is only 0,382 or 38,2%. CA and CE together make an average predictor for SumNCSS, therefore Hypothesis 2 holds true for SumNCSS:

Sum

The multiple linear regression with Sum as dependent and CE, CA and A as independent variables is shown in Table 39:

Model Summary[d]

Model	R	R Square	Adjusted R Square	Std. Error of the Estimate
1	,370[a]	,137	,137	1,22320
2	,418[b]	,175	,174	1,19644
3	,419[c]	,175	,174	1,19633

a. Predictors: (Constant), LogCe

b. Predictors: (Constant), LogCe, LogCa

c. Predictors: (Constant), LogCe, LogCa, LogA

d. Dependent Variable: LogSum

Table 39: Multiple linear Regression Sum (dependent variable) and CE, CA and Abstractness (independent variables)

The increase in predictive capability to using only CE as predictor is similar to SumNCSS. After the results in the linear regression analysis this was expected. Looking at the coefficients reveals a clearer picture:

Coefficients[a]

Model		Unstandardized Coefficients		Standardized Coefficients	t	Sig.
		B	Std. Error	Beta		
1	(Constant)	,353	,084		4,179	,000
	LogCe	1,644	,087	,370	18,874	,000
2	(Constant)	,276	,083		3,329	,001
	LogCe	1,428	,088	,322	16,263	,000
	LogCa	,613	,061	,200	10,117	,000
3	(Constant)	,251	,086		2,931	,003
	LogCe	1,447	,089	,326	16,214	,000
	LogCa	,587	,064	,192	9,133	,000
	LogA	,446	,374	,024	1,192	,233

a. Dependent Variable: LogSum

Table 40: Coefficients of multiple linear regression for Sum

Abstractness and the constant factor have to be discarded as insignificant. CA contributes more to Sum than it did to SumNCSS, when looking at the t-values. Still CE accounts for the major part of the variance in Sum. Hypothesis 2 does not hold true for Sum as the portion of the variance explained by the set of CA and CE fails to account for a sensible value.

SPSS knowledge was acquired by using [BüAc06] and [FiAn05].

7 Discussion

7.1 General

The first thing to mention, as it likely had a considerable impact on this study, is the quality of the CVS data acquired from SourceForge. As can be derived from the amount of projects that could not be worked with, CVS usage varies greatly from project to project (and probably user to user). Before going into details on the results, it needs to be pointed out that this might be the reason of the average performance of the structure metrics. Another problem that had to be dealt with was the “wrong” usage of CVS, namely making hard copies of versions into the same directory. The metrics could not distinguish between the directories and the effort of going through each project manually, working the parsers on each file, renaming, etc. would have seriously crippled the automated process that allowed for the huge number of classes to be analysed. Furthermore, when the CVS was used like that the Sum data was obviously wrong, as the Sum from previous versions should be included in the final Sum.

Therefore only projects that had an insignificant (depending on project size, between 50 and 100 as maximum) amount of double entries, etc. were used in the study. While this almost halved the number of projects aimed for in the beginning of the work, 18.000 classes and more than 2.000 packages analysed is a huge amount of data to work with. There was a small field study performed in the beginning of this work (which is not documented here) of ten projects that did not encounter these problems, so this erratic use of CVS (erratic for the CVS data to be used for this work) came unexpected. This does not imply that the usage of CVS on SourceForge per se is done erroneous; those projects just do not deliver data that can be used in this study.

The data gathered from the CVS is not normally distributed. Specifically Sum, due to a lot of classes not having any changes reported, is heavily positively skewed. Even after taking the logarithm, it still showed a skewed histogram. The histogram for SumNCSS after taking the logarithm strongly resembled a normal distribution however. This is an indicator that SumNCSS is a better measure than Sum alone as it is more finely grained (Sum has a huge amount of zero values rendering it more coarse) and if size is treated as a one-time addition and therefore as a change, it is in the spirit of maintenance effort as well. Unfortunately with Sum one third of the data gathered were cases of zero changed files. If that data would be eliminated, it would bias the outgoing of the study too much. So it was decided to present SumNCSS and Sum, but when looking at the lower results of Sum, it is necessary to keep in mind that the large amount of zero values skews Pearson’s correlation.

Before the code metrics and the results are interpreted, it is necessary to analyse the relationship of Sum and NCSS since Sum and the addition of Sum and NCSS have been used as dependent variables for the regression analysis. As expected Sum and NCSS correlated; a low positive correlation could be seen. NCSS is the most basic predictor for Sum, following common sense that a class consisting of many lines of code will have a high number of changes. This crude measure is reflected by a low R Square value and a low Pearson-correlation. The low value partly stems from the zero values (eliminating the zero values shows an average correlation of 0,521); other reasons are generated code (GUI designers, Model-Driven Development) and various more. On package level the results of the regression analysis were more convincing, correlation was average with NCSS covering 27,4% of the variance in Sum. This is closer to the expected relationship of Sum and NCSS.

As a first step, Code Metrics will be summed up then a conclusion on the Structure Metrics will be drawn. Finally the results of the multiple regressions for both kinds of metrics will be evaluated. For categorizing the quality of a correlation, the categories from [BüAc06] will be used:

- Less or equal than 0,2 → very low Correlation
- Less or equal than 0,5 → low Correlation
- Less or equal than 0,7 → average Correlation
- Less or equal than 0,9 → high Correlation
- More than 0,9 → very high Correlation

The last part of the Conclusion will deal with the results of related publications (some of which have been mentioned in earlier Chapters).

7.2 Code Metrics

7.2.1 General

In general, the code metrics performed better than the structure metrics. This might be the case because they interpret less; it seems the more sophisticated a metric (the exception being WMC, but only on first sight), the less of a predictive quality it holds. The following Hypotheses have been examined for the code metrics:

Hypothesis 1.4: A high value of WMC of a class will lead to a high value of changed lines. This is based on the assumption that complexity increases amount of change done.

Hypothesis 1.5: A high value of LCOM* will lead to a high value of changed lines. A low value on LCOM* suggests good design of classes, while a high value hints at too many tasks performed by a class. A bad design of a class should lead to an increase in changed lines.

Hypothesis 1.7: A high value of NOM suggests much functionality embedded in a class and therefore high complexity in the class. Opposed to that is the concept of encapsulation where a high number of methods implies low complexity.

Hypothesis 1.7.1: NOM and WMC should have significantly different results as the complexity factor measured in WMC should discard the getter/setter complexity that is skewing NOM.

The Hypotheses will be evaluated in the according chapters followed by a summary of the performance of the Code Metrics.

7.2.2 Number Of Methods

*“**Hypothesis 1.7:** A high value of NOM suggests much functionality embedded in a class and therefore high complexity in the class. Opposed to that is the concept of encapsulation where a high number of methods implies low complexity.”*

The NOM metric should have positive correlations with SumNCSS and Sum according to Hypothesis 1.7. Furthermore the correlation should be of an average value. This is the case with SumNCSS. The correlations in detail of 0,641 with SumNCSS and 0,338 with Sum suggest an average and a low connection of NOM with the tested dependent variables. Since it is a positive correlation and significant, Hypothesis 1.7 holds true. However the R Square value for Sum is too low to warrant any reliability for NOM as predictor for Sum. The R Square value of 0,411 for SumNCSS indicates that NOM is an average predictor for SumNCSS. The lower performance of Sum was expected due to the skewed nature of the data. Furthermore, the average correlation value of SumNCSS and NOM imply that Hypothesis 1.7 holds true for the data used in this study. The average value was expected due to the nature of the object oriented paradigm that conflicts with the idea behind NOM. With 41,1% of the variance in SumNCSS covered by NOM, NOM makes for a rather good predictor for SumNCSS, while the low value of 11,4% variance-coverage with Sum makes NOM not viable for Sum.

The R Square value of SumNCSS and NOM is considered “good” in this study, because it is actually higher than expected from Hypothesis 1.7.

7.2.3 Weighted Methods per Class

***“Hypothesis 1.4:** A high value of WMC of a class will lead to a high value of changed lines. This is based on the assumption that complexity increases amount of change done.”*

As with NOM, WMC shows a big discrepancy in correlations between SumNCSS and Sum (0,773 for SumNCSS as opposed to 0,420 for Sum). This leads to a very good R Square value for SumNCSS and a low R Square value for Sum. The performance of WMC is overall better than NOM and suggests Hypothesis 1.7.1 holds true, as well as Hypothesis 1.4. WMC covers 59,8% of the variance in SumNCSS. This makes WMC a very good predictor for SumNCSS, while 17,6% coverage with Sum shows the same conclusion as with NOM apply. WMC is a better predictor than NOM for both dependent variables and a trend can be perceived with both variables as well. Correlation is positive and there is a linear relationship of WMC and the dependent variables.

If simple percentages of predicting qualities are used (R Square values as percentages), WMC is roughly 20% (41,1% vs 59,8%) more efficient than NOM. This is however a very crude assumption; more accurate is that WMC is a better predictor than NOM for the data gathered in this study. This is mostly due to the nature of WMC and its independence regarding programming paradigms, while NOM is not. The complexity measure (McCabe’s cyclomatic number) itself is implemented as a basic count of certain operators in the program. A correlation this high with NCSS is worthy of another examination in a different study.

While WMC seems to be a rather sophisticated metric, a closer look reveals that it is not. It sums up the operators listed in Chapter 4 of each method, which are then added up for each class. And this might be the reason why the correlation of NCSS and WMC is that high, since those operators make up most of the code in a program, except for Getter and Setter methods. Thus this correlation (0,936 NCSS/WMC correlation) suggests that only a small part of classes are actually encapsulation objects. This needs however further examination and can be only assumed at this point.

Hypothesis 1.7.1 states that WMC should perform better than NOM and the data seems to carry this Hypothesis. All correlations and regression analyses are of a higher value with WMC. The relationship of NOM and WMC alone could fill a scientific paper, but is not a point of interest in this study; it would be interesting to know how much of NOM is represented in WMC for instance. Part of this topic will be briefly touched in the multiple linear regression model analysis, without going into the required depth of a thorough examination.

A remark as to why WMC performed poorly for Sum. If the zero values are removed for Sum, the R Square values rise to 0,274. This is a rather big improvement and further indicates that the zero values skew Sum. However this is not valid for this study as the interest lies in all classes and cutting out one third of the data can hardly be deemed representative.

7.2.4 Lack of Cohesion Of Methods

*“**Hypothesis 1.5:** A high value of LCOM* will lead to a high value of changed lines. A low value on LCOM* suggests good design of classes, while a high value hints at too many tasks performed by a class. A bad design of a class should lead to an increase in changed lines.”*

Of the three code metrics, LCOM* had the lowest correlations (and therefore the lowest R Square values). LCOM* shows low correlations with SumNCSS and Sum, with SumNCSS on the upper end of low correlations (0,464) and Sum at the lower end of low correlations (0,259). LCOM* covers 21,5% of the variance in SumNCSS and 6,7% of the variance in Sum. Since the correlation is definitely positive for both dependent variables, Hypothesis 1.5 holds true. However LCOM* does not make a good predictor for both dependent variables.

This result was already anticipated in the theoretical part of this work. LCOM* measures grouping of similar tasks by counting out shared variables (simply put). The implication is that a low cohesion is an indicator for bad design and therefore increased complexity. LCOM* measures the absence of cohesion, therefore a high value of LCOM* implies a high value of changed lines. LCOM* “penalizes” the use of getters and setters, therefore skewing the results of the examination. The low predictive quality of LCOM* therefore suggest a high usage of getters and setters, which is directly countered by the high correlation of WMC and NCSS. Since LCOM* is a rather troubled metric and subject to frequent changes in its implementation, this study is inclined to value the score of WMC more, as it is less sophisticated than LCOM* and therefore less error-prone. This is an assumption based on interpreting the results born of the assembled data however and do not imply a general state.

7.2.5 Summary: Code Metrics

All Hypothesis about the nature of the relationship of dependent variables and Code Metrics held true. There were significant and positive correlations for all three of the metrics. Furthermore the expected ranking of the metrics expressed in Hypothesis 1.7.1 held true as well. The gathered data supports these assumptions. The low score of LCOM* was anticipated as well, as has been indicated in Chapter 4. LCOM* has undergone significant changes and on top of that it penalizes the usage of encapsulation objects. This is reflected by the low scores of course. SumNCSS proved to be the “better” dependent variable, as the metrics were able to predict it much better. This stems from the fact that all three code metrics have a higher correlation with NCSS than with Sum and therefore SumNCSS. However NCSS is not of interest for this study as it is not an indicator of effort needed to be put into certain classes during development and maintenance. Sum suffers heavily from the 30% of zero values found in the CVS and a conclusion from the high number of zero values would be that a high number of getters and setters were used. This assumption is negated by the very high correlation of NCSS and WMC. At this point of the study it is impossible to tell whether a high number of setters and getters was used (zero values in change history, low LCOM* value) or the discrepancy stems from using pre-generated classes, code reuse, etc. A whole set of studies could be undertaken in this area.

This does not change the fact though that WMC is the best predictor for both Sum and SumNCSS in the gathered data followed by NOM and LCOM* on the last place. All results are significant at $p < 0{,}001$.

The next step of the analysis is to look into Hypothesis 2 for the Code Metrics. This will be done in the multiple regression analysis, after going through the structure metrics.

7.3 Structure Metrics

7.3.1 General

The structure metrics were executed on package level, with the exception of the DIT metric, which was performed on class level. As mentioned earlier, the performance of the structure metrics was worse than the performance of the code metrics. Considering the better NCSS/Sum correlation on package level, this seems irritating. Why the metrics performed in the fashion they did will be examined in this chapter. The following Hypotheses were tested:

Hypothesis 1.1: A high value of Efferent Coupling of a package will lead to a high value of changed lines. This is based on the assumption that instable packages are changed easier and therefore are changed more often [MaRo03].

Hypothesis 1.2: A high value of Afferent Coupling of a package will lead to a low value of changed lines. According to [MaRo03] a stable package is harder to change and therefore will change less.

Hypothesis 1.3: A high value of Abstractness suggests a stable package (refer to the earlier part of this chapter). Therefore a high value of Abstractness should lead to a low value of changed lines.

Hypothesis 1.3.1: A high value of Abstractness suggests a stable package and should therefore have a high value of Afferent Coupling. Refer to appendix for chart.

Hypothesis 1.6: A high value of DIT for a class suggests a class that inherits a lot of methods and therefore is of greater design complexity. A high value of DIT of a class should lead to increased changed lines.

7.3.2 Efferent and Afferent Coupling

*"**Hypothesis 1.1:** A high value of Efferent Coupling of a package will lead to a high value of changed lines. This is based on the assumption that instable packages are changed easier and therefore are changed more often [MaRo03]."*

If Hypothesis 1.1 holds true for the analyzed data, a positive correlation is expected. With an average correlation of 0,585 with SumNCSS and a low correlation of 0,370 with Sum, Hypothesis 1.1 holds true. However with only an average R Square of 0,342 (34,2% of the variance in SumNCSS covered by Efferent Coupling) with SumNCSS and 0,137 (13,7% of the variance in Sum covered by Efferent Coupling), Efferent Coupling is only of average predictive quality.

This average performance of Efferent Coupling was not expected as the data on package level seemed "better"; that is Sum and NCSS were correlating better. If the analysis of Efferent Coupling was a slight letdown, the analysis for Afferent Coupling was staggering.

*"**Hypothesis 1.2:** A high value of Afferent Coupling of a package will lead to a low value of changed lines. According to [MaRo03] a stable package is harder to change and therefore will change less."*

While Efferent Coupling presents the number of classes inside a package depending on outside classes (representing an instable package), Afferent Coupling presents the number of classes outside of a package depending on classes inside of the package, therefore suggesting a stable package (See Section 4.3 for an in-depth explanation). Basically an opposing trend to Efferent Coupling was expected. After calculating the correlations, a significant albeit low positive correlation was found. A stable package is therefore more likely to change than not. This is curious and completely opposite to the Hypothesis 1.2 and an explanation is hard to find. Basically the data gathered for this study supports the concept of instability as propagated in Section 4.3 while not supporting the opposite of this concept. The exact correlations are 0,336 with SumNCSS and 0,278 with Sum.

The data gathered in this study therefore supports Hypothesis 1.1, but negates Hypothesis 1.2 at a significance level of p < 0,001. Afferent Coupling shows an R Square value of 0,113 with SumNCSS and 0,077 with Sum, which means a low predictive quality for both.

7.3.3 Depth-of-Inheritance Tree

*“**Hypothesis 1.6:** A high value of DIT for a class suggests a class that inherits a lot of methods and therefore is of greater design complexity. A high value of DIT of a class should lead to increased changed lines.”*

Hypothesis 1.6 suggests a positive connection of DIT and SumNCSS/Sum. With correlations of 0,149 and 0,115 significant with p < 0,001, this is the case. However it is a very low positive correlation (Figure 21 shows a very flat regression line), suggesting that the predictive quality of DIT will be close to nil. Actually DIT shows R Square values of 0,022 with SumNCSS and 0,013 with Sum and which is not enough to be able to predict anything by using DIT and the linear regression model. One of the reasons for the bad performance of DIT may be that the linear regression model is just not able to fit the data, however from the scatterplots of the residuals and the predicted values (in Figure 22) and DIT and SumNCSS (in Figure 21) no pattern emerges which would actually fit the data better.

Another reason for the bad performance of DIT might be found in the implementation of the metric itself. If a class has several inner classes, the DIT position is the summed up class value. However if only the classes without inner classes are examined (~15000 classes) the correlation of DIT and SumNCSS and Sum actually becomes insignificant. If only the classes with inner classes are analyzed, the correlation is low, but significant again. If the SumNCSS and Sum values are divided by the number of classes and correlated, significant correlations slightly above zero emerge. No pattern can be deducted from this though. Basically the last few steps should provide insight how the value of DIT was generated; Manipulating the data and splitting it up won’t change the fact that DIT does not deliver any predictive value or proves Hypothesis 1.6 for that matter. With values this close to zero Hypothesis 1.6 has to be discarded or at least handled sceptically.

7.3.4 Abstractness

***“Hypothesis 1.3:** A high value of Abstractness suggests a stable package. Therefore a high value of Abstractness should lead to a low value of changed lines.”*

Abstractness uses a similar premise as Afferent Coupling. The concept of a stable package is used which is reasoned to be an indicator for a piece of code that is not likely to change. This is reflected by Hypothesis 1.3. With the results from Afferent Coupling, Abstractness should be an indicator whether the concept of stable packages is applicable or not. With a correlation value of 0,052 with Sum and 0,021 with SumNCSS the implication that the concept of stability is not fitting to the change data gathered in this study is strong. Again where a negative correlation was expected, a very low positive correlation is found (except for NCSS and Abstractness which does show a negative correlation). However opposed to Afferent Coupling, these correlations (including NCSS) are NOT significant at the threshold chosen for this study (p < 0,001). Hypothesis 1.3 has to be discarded as no correlation between metric and dependent variable could be found. Following the insignificant correlations, regression analysis yields no results.

After gathering these results, the expectations for the analysis of Hypothesis 1.3.1 were small:

***“Hypothesis 1.3.1:** A high value of Abstractness suggests a stable package and should therefore have a high value of Afferent Coupling. Refer to appendix for chart.”*

Since both Abstractness as well as Afferent Coupling measure stable packages, they should have a positive correlation (if the concept of a stable package is applicable) with each other. This Hypothesis holds true as the correlation between the metrics is 0,301. With R Square of 0,091 Abstractness cannot be used as a predictor for Afferent Coupling.

7.3.5 Summary: Structure Metrics

After analysing the relationship of Sum and NCSS for packages, it was expected that the structure metrics should actually yield better results than the Code Metrics (except DIT of course, which was performed on class level). While Efferent Coupling proved to be of more predictive power than LCOM*, the weakest Code Metric, the other metrics failed partly. DIT showed the expected tendency of a positive correlation; however the values were too low to be able to use it as predictor. The concept that a stable package is unlikely to change is not carried by the data assembled in this study. A stable package identified by Afferent Coupling and Abstractness is actually more likely to change than not (to be more precise: the package will have a random number of changes unrelated to Afferent Coupling or Abstractness). However the concept of the unstable package does hold true as it is implemented by Efferent Coupling. While this seems like a conflict at first glance, when looking at the way the metrics work it does make sense that it is possible that one metric is delivering good results while the others do not.

Efferent Coupling measures the numbers of classes inside a package depending on classes outside the package. Afferent Coupling measures the numbers of classes outside a package depending on classes inside of a package. So Efferent Coupling and Afferent Coupling actually measure independent items. The error therefore is more likely to be found in the definition of a stable package and by the measurement of it. Abstractness on the other hand is per definition an accompaniment of stability. It is the number of abstract classes of a package divided by the number of all classes. If a lot of abstract classes are found in a package, it means a lot of packages will depend on it, thus discouraging change in this package, since any changes done here will affect numerous classes. This is basically the definition of stable used for Afferent Coupling. However the data does not reflect this behaviour. Obviously the change for a stable package varies so much that it is not possible to come to the conclusion given in literature [MaRo03] by means of the data gathered in this study.

As with the Code Metrics, the Structure Metrics were better predictors for SumNCSS than for Sum; the exception being Abstractness, which is negligible since the correlations themselves are not significant enough for this study. This is of course due to the already discussed nature of the data acquired from the CVS.

After discussing the implications of each metric, the results for the multiple regression analysis will be discussed.

7.4 Stepwise Multiple Regression Analysis

7.4.1 General

Multiple regression analysis was used to find out whether using a set of metrics will make for a better predictor than using a single metric. Since most of the Structure Metrics operate on package level, two analyses had to be done; one on class level, using the Code Metrics and DIT and one on package level using Efferent and Afferent Coupling and Abstractness. These analyses were used to prove (or discard) Hypothesis 2:

***"Hypothesis 2:** If Hypothesis 1 holds true, is there a set of metrics that is valid in all projects under examination?"*

Since Hypothesis 1 held true for all Code Metrics and for DIT as well, the results of the analysis of the Code Metrics were expected to be positive as well. However Hypothesis

1 did not held true for two out of the three metrics used on package level, so the expectations for viable results for these were low.

7.4.2 Multiple Regression Analysis on class level

The metrics were entered by their correlations, starting with WMC having the highest value going to DIT with the lowest value.

SumNCSS

The improvement from only WMC to using all four metrics is minimal (an increase of 0,003 in correlation and 0,005 in R Square) yet all metrics are significant at p < 0,001 suggesting that they should be included in the regression formula. The t-values for the metrics are similar to this, as WMC has the highest by far, which means that it contributes the most. This also suggests that WMC covers almost all predictive quality in the other three metrics, except for those 0,5% added. Yet for the best fit of the model all four are required. This means Hypothesis 2 holds true for the four metrics, the set is valid, since it is significant for all projects under examination. Due to the major role of WMC, using just this metric singularly seems sufficient for practical use.

Sum

The improvement for Sum of using four metrics as predictors is bigger than with SumNCSS for the correlation, yet the pattern remains the same. The increases are 0,007 in correlation and 0,005 in R Square. WMC covers most of the predictive power of the model, yet all factors are significant at p < 0,001 for the regression formula. Hypothesis 2 holds true for Sum.

7.4.3 Multiple Regression Analysis on package level

As with the Code Metrics, the metrics were entered ordered by their correlations; that is first metric was Efferent Coupling, followed by Afferent Coupling and Abstractness as last metric. As expected, Abstractness was not significant for either Sum or SumNCSS.

SumNCSS

Efferent Coupling is showing the biggest correlation with SumNCSS, when adding Afferent Coupling the correlation of the model rises by 0,033. This is a bigger improvement than the improvement of all three Code Metrics added to WMC together. The predictive power grows by 0,039 (3,9% more of the variance in SumNCSS are explained by this model). Abstractness, as indicated earlier, is not significant for this model. With a correlation of 0,618 and 38,2% of the variance in SumNCSS covered by this new model, it is actually viable to use both metrics (even though Afferent Coupling does not measure what it is supposed to). The gain in predictive power by using both metrics is definitely notable and Hypothesis 2 holds true for using the set of Efferent and Afferent Coupling to predict SumNCSS.

Sum

Using Efferent and Afferent Coupling on Sum reveals a pattern already shown by the class level. The gains of predictive power of the model are about the same; correlation increases by 0,048 and R Square by 0,037. The same conclusion for Hypothesis 2 can be drawn from the analysis. It is better to use both Efferent and Afferent Coupling. Worth mentioning is that the constant factor when using stepwise multiple regression analysis becomes insignificant as well.

7.5 Comparison with related publications

The first study of interest to look at is [LiHe93], as it uses the same measurements as used in this study. One of the most obvious differences is the handling of the data. Whether or not the data in [LiHe93] has been logarithmized cannot be found out at this point; looking at the distribution of their data shows exponential data. There are 110 classes collected from two projects. The Kolomogorov-Smirnof-Test for the change data (after logarithmizing it) is insignificant (0,140); the change data is normally distributed. There are no zero-change classes in the study. This is due to the nature of the projects. Both projects are commercial products and therefore are prone to stricter code (and change data) monitoring. This leads to less “noise” (the zero classes) in the data. However it is apparent that two projects from the same developer can hardly be representative. Code handling, developing routines and organization in general introduce bias. The analysis done in [LiHe93] might be representative for the Software Developer that the projects originate from; it is hardly representative for the appliance of metrics on general projects, even less on projects with a more lax handling of change data tracking that are common in the Open Source Community.

Due to the restrictions in [LiHe93], the results of their analyses were overall better than the results found in this study. There is a set of 8 metrics introduced that delivers R-Square values of 0,9030 and 0,8680 (the analysis is done for each of their projects). This is overall better than the result of the set of WMC, NOM, DIT and LCOM* in this study; since there was no single metric analysis undertaken, it is hard to determine if the trends for the single metrics are the same. The general trend is similar though (strong positive correlation and a good predictive quality). Abstractness, Efferent and Afferent Coupling were not introduced in [LiHe93].

Studies like [GyFe05] that were analysing fault-rates and related parameters are hard to compare as, due to the nature of their data, these studies are using logistic regression analysis. These studies are predicting if a class has any faults or no faults at all, while this work uses linear regression. The study in [GyFe05] is remarkable because it covered 8.000 classes, so it is of significant size. However it is not clear whether the data used in the study were logarithmized or not. Logistic Regression assumes normal distribution as well. The second part of [GyFe05] runs a linear regression analysis on the metrics and reveals trends similar to this work. WMC performed better than LCOM and both perform better than DIT. The low results for correlations are explained, as Spearman's correlation should have been used instead of Pearson's. This would very likely have netted better results for the correlations, which are all below average or of lower level. Furthermore a requirement for linear regression, namely normally distributed data, was not met if the data was not logarithmized. To further fit the data, the data should have either been logarithmized or non-parametric regression should have been used.

One of the reference studies for [GyFe05], [BaBr96] uses 180 classes for a similar analysis. While [GyFe05] uses a more than sufficient sample size and therefore reveals similar trends to this study, [BaBr96] uses only 180 classes and suffers from the low sample size, as only 36% of the classes actually contained faults and 86% of these classes contained less than three faults. This renders any statement about the metrics useless. Furthermore the already low sample is divided into three categories, which only decreases the sample and power of their study. They end up with R Square values of 0,007 for the dependent variable faulty classes by independent variable WMC. Another curious note worth mentioning is that obviously a p-value for the study was not set in advance of the study. This is bad style according to [FiAn05]. It is therefore impossible to compare [BaBr96] with this study (even though trends are similar from [BaBr96] to [GyFe05] in the logistic regression part).

[HeKa81] examined 165 procedures in a Unix system. The study correlated metrics between themselves and with errors reported. The study used McCabes and Halsteads and achieved very good results with their error data. However, not enough information can be found on the statistical methodology used in [HeKa81] to effectively compare their work with this study. Furthermore only McCabe's Cyclomatic Complexity was used as part of WMC in this study and detailed results for only McCabe's per function have not been examined. Additionally this study deals with the Object-Oriented Paradigm, while [HeKa81] does not.

[KaCa85] uses an approach completely different from this work. This study does not use regression analysis, but uses a threshold of the chosen dependent variables and the independent variables. This threshold is the doubled standard deviation. The study analyzes whether or not the independent value is out of the threshold if the dependent value is outside of that threshold. The reason for this style of examination is that they find regression analysis not suited for metric data as previous studies only yielded low results and therefore discouraging the use of regression analysis. One of the reasons for the low results in previous studies might have been the unsatisfactory usage of regression analysis though. [GyFe05] and [BaBr96] show exponential distribution patterns of their raw data. Whether or not the data has been logarithmized cannot be learned from the studies. If it was not logarithmized they breached one of the requirements for regression analysis, namely normally distributed data.

While [KaCa85] brings a new, interesting point of view into play, the threshold system has several drawbacks. The first one is that the threshold is defined by the measured data and therefore it is not possible to globally define it (It can only be gathered after measuring the data you want to predict). Secondly using statistical outliers as base of an examination seems unorthodox. Thirdly there is no granulation; either a case is outside the threshold or within. The distance from that threshold does not play a role in the analysis. However [KaCa85] is about the validation of metrics and not about metrics as predictors, therefore their approach is useable for their study. The approach is not useable in case of the study performed in this work however.

[YuSy02] run a similar analysis to [GyFe05] with a few different metrics. However [YuSu02], [GyFe05] and this work share a set of metrics and these metrics deliver the same ranking of predictive quality discovered in this work, namely NOM, followed by LCOM, followed by DIT. This fascinating parallel indicates a relationship between Changed Lines and Fault-Proneness. The reason for this similarity might stem from the fact that the data in [YuSy02] was normalized; further information on the statistical methods used is not given however, making yet again a direct comparison of the results impossible.

Due to the unique approach of this work it is hard to compare the results gathered from this study with existing studies. As stated in Chapter 1, most metrics are validated against failure data. Additionally to the different nature of the other studies, those other studies use a different statistical way of analyzing their data. Logistic regression is used instead of linear regression. Furthermore there is not enough information given on the way the data was handled in the studies. There are distributions shown in some studies ([GyFe05] and [BaBr96]), those are not normally distributed, yet regression was applied on that data. Whether the data was logarithmized (to achieve normal distribution) or not can only be assumed at this point.

The [LiHe93] study shows a similar trend than the study in this work, but again it is impossible to say whether the data of this study was transformed or not. The tests given in this work were not performed in [LiHe93], but by the author of this work with data catalogues given in the appendix of [LiHe93].

To make assumptions about connections between the studies presented in this last part and the study in this work can only be done in a crude fashion. Basically it is impossible to state more than:

- The ranking of the metrics is mostly the same as in this study
- If the data was transformed (and this transformation was commented in the study), the results are closer to this study
- Fault-Proneness and Change Data seem to be connected
- Change Data performs better than Fault Data
- It is easier to work with Change Data than with Fault Data

Furthermore the programming languages were different in the other studies ([GyFe05] uses C++ e.g.).

8 Conclusion

The analyzed Software Metrics show a definite relation between most of the metric-produced values and the measured change data. The Code Metrics provide overall better results than the Structure Metrics. Of the Code Metrics WMC and NOM deliver the best results and show a good correlation with Change Data. The more complex the metrics are, i.e. the more concepts they try to realize, the less of a correlation can be found between metrics and Change Data. This can be seen especially in the analysis of the Structure Metrics. However, while the Structure Metrics used in a stand-alone fashion generally fail to deliver strong correlations (except one metric), a set of Structure Metrics could be found that showed an average correlation value (and an average R- Square value) for Change Data. Considering the fact that these metrics can be applied early in the development of a project (only imports are basically required for the coupling metrics), this is a favourable result for the Structure Metrics.

A concept introduced in literature, the concept of Stability [MaRo03], has to be discarded for this study. The results delivered by the regression analysis are completely opposite of this concept. Part of this failure of the Stability concept is the Abstractness metric. Out of all the examined metrics only Abstractness was not able to deliver significant results with Sum and SumNCSS.

Out of the two analyzed dependent variables, the metrics showed a higher correlation with SumNCSS than with Sum. The reasons for this can be found in the CVS usage of the SourceForge-Community. Out of 100 projects originally aimed for, only 46 could be finally used for various reasons. In the final study 18000 classes and over 2000 packages were analyzed.

For future studies it will be of interest to examine the relationship of the metrics introduced in this study with Change Data in the future. This study examined the status of Change Data and metrics at the same timeframe (backward approach) – the capability of the metrics to provide future Change Data still needs to be examined (forward approach).

Bibliography

[KaSt02] S. Kan: Metrics and Models in Software Quality Engineering, Second Edition. Addison Wesley, 2002

[BaBr96] V. R. Basili, L.C. Briand, W.L. Melo: A Validation of Object-Oriented Design Metrics as Quality Indicators, *IEEE Trans. Softw. Eng.* 22, 10 (Oct. 1996), 751-761.

[KaCa85] D. Kafura, J. Canning: A validation of software metrics using many metrics and two resources. In *Proceedings of the 8th international Conference on Software Engineering* (London, England, August 28 - 30, 1985). International Conference on Software Engineering. IEEE Computer Society Press, Los Alamitos, CA, 378-385.

[HeKa81] S. Henry, D. Kafura, K. Harris: On the relationships among three software metrics. In *Proceedings of the 1981 ACM Workshop/Symposium on Measurement and Evaluation of Software Quality* H. J. Highland, Ed. ACM, New York, NY, 81-88. 1981.

[NaBa06] N. Nagappan, T. Ball, A. Zeller: Mining metrics to predict component failures. In *Proceeding of the 28th international Conference on Software Engineering* (Shanghai, China, May 20 - 28, 2006). ICSE '06. ACM, New York, NY, 452-461.

[MeSu99] M. Mendonca, N.L. Sunderhaft: Mining Software Engineering Data: A Survey. A DACS State-of-the-Art Report, Survey for Air Force Research Laboratory Information Directorate, 525 Brooks Road Rome, NY 13441-4505, 1999.

[ChKe91] S.R. Chidamber, C.F. Kemerer: Towards a metrics suite for object oriented design. In *Conference Proceedings on Object-Oriented Programming Systems, Languages, and Applications* (Phoenix, Arizona, United States, October 06 - 11, 1991). A. Paepcke, Ed. OOPSLA '91. ACM Press, New York, NY, 197-211.

[KaDe85] D. Kafura: A survey of software metrics. In *Proceedings of the 1985 ACM Annual Conference on the Range of Computing : Mid-80's Perspective: Mid-80's Perspective* (Denver, Colorado, United States). ACM '85. ACM, New York, NY, 502-506.

[HeSe96] B. Henderson-Sellers: Object-Oriented Metrics: measures of complexity; Prentice Hall PTR, Upper Saddle River, New Jersey 07458 1996

[BoGu06] S. Bouktif, Y.G. Gueheneuc, G. Antoniol: Extracting Change-patterns from CVS Repositories, *Reverse Engineering, 2006. WCRE '06. 13th Working Conference on* , vol., no., pp.221-230, Oct. 2006

[LoM93] M. Lorenz: Object-Oriented Software Development: A Practical Guide. Prentice Hall, New Jersey, 227 pp. 2003

[LiHe93] W. Li, S. Henry: Object-oriented metrics that predict maintainability. *J. Syst. Softw*. 23, 2 (Nov. 1993), 111-122.

[MaRo03] R. Martin: Agile Software Development – Principles, Patterns, and Practices. 2003 Pearson Education, Upper Saddle River, New Jersey 07458

[ChKe91] S.R. Chidamber, C.F. Kemerer: Towards a metrics suite for object oriented design. In *Conference Proceedings on Object-Oriented Programming Systems, Languages, and Applications* (Phoenix, Arizona, United States, October 06 - 11, 1991). A. Paepcke, Ed. OOPSLA '91. ACM, New York, NY, 197-211. .

[CoHo05] M. Conklin, J. Howison, K. Crowston: Collaboration using OSSmole: a repository of FLOSS data and analyses. In *Proceedings of the 2005 international Workshop on Mining Software Repositories* (St. Louis, Missouri, May 17 - 17, 2005). MSR '05. ACM, New York, NY, 1-5.

[SoFo] SourceForge, Open Source software development web site.
http://sourceforge.net/, 12.03.2008

[OsSo] FLOSSmole, Project Page.
http://ossmole.sourceforge.net/, 12.03.2008

[JaNc] JavaNCSS, Project Page.
http://www.kclee.de/clemens/java/javancss/, 12.03.2008

[MySq] MySQL, Project Page.
http://www.mysql.com/, 12.03.2008

[SoKu] Krugle Search Engine for SourceForge, Search Engine Homepage.
http://sourceforge.krugle.com

[WeDa05] D. Weiss: A Large Crawl and Quantitative Analysis of Open Source Projects Hosted on SourceForge. Institute of Computing Science, Poznań University of Technology, Poland, Research Report RA-001/05, 2005

[NoDa] University of Notre Dame, SourceForge.net Research Data. http://www.nd.edu/~oss/Data/data.html, 12.03.2008

[SoDo] SourceForge Statistics, DocumentD04. http://sourceforge.net/docs/D04/en/, 12.03.2008

[SoMe] Metrics 1.3.6, Project page. http://metrics.sourceforge.net/, 12.03.2008

[ClJd] JDepend, Project Page. http://www.clarkware.com/software/JDepend.html, 12.03.2008

[StPa] SPSS Homepage. http://www.spss.com/, 12.03.2008

[McTh76] T. McCabe: A complexity measure. In *Proceedings of the 2nd international Conference on Software Engineering* (San Francisco, California, United States, October 13 - 15, 1976). International Conference on Software Engineering. IEEE Computer Society Press, Los Alamitos, CA, 407.

[WaMc96] A. Watson, T. McCabe: Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric. Computer-Systems Laboratory National Institute of Standards and TechnologyGaithersburg, MD 20899-0001, August 1996

[ChKe94] S.R. Chidamber, C.F. Kemerer: A Metrics Suite for Object Oriented Design. *IEEE Trans. Softw. Eng.* 20, 6 (Jun. 1994), 476-493.

[BüAc06] A. Bühl: SPSS 14 – Einführung in die modern Datenanalyse. Pearson Studium, Martin-Kollar-Straße 10-12, D-81829 München/Germany 2006

[FiAn05] A. Field: Discovering Statistics using SPSS. Sage Publications Ltd, 1 Oliver’s Yard, 55 City Road, London EC1Y 1SP 2005.

[GyFe05] T. Gyimothy, R. Ferenc, I. Siket: Empirical Validation of Object-Oriented Metrics on Open Source Software for Fault Prediction. *IEEE Trans. Softw. Eng.* 31, 10 (Oct. 2005), 897-910.

[YuSy02] P. Yu, T. Systa, H. Müller: Predicting Fault-Proneness Using OO Metrics: An Industrial Case Study. In *Proc. Sixth European Conf. Software Maintenance and Reeng.* (CSMR 2002), pp. 99-107, Mar. 2002.

Appendix

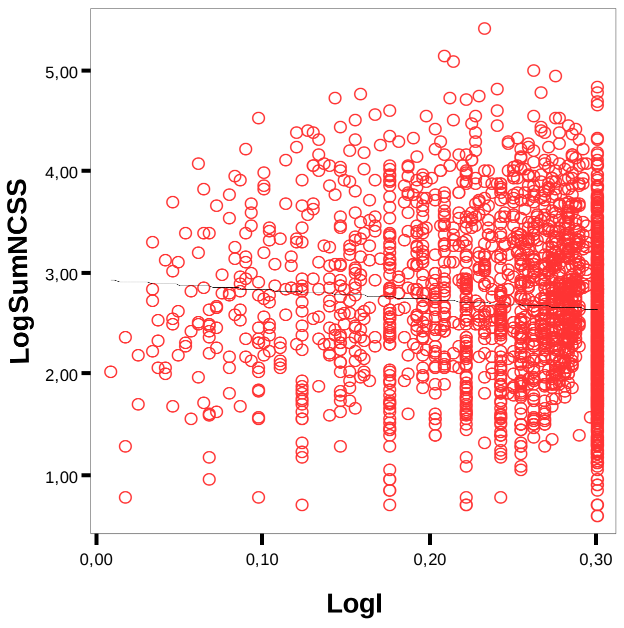

Figure 27: Scatterplot of SumNCSS and Instability

Correlations

		LogSum	LogNCSS	LogSum NCSS
LogI	Pearson Correlation	-,102**	-,029	-,079**
	Sig. (1-tailed)	,000	,084	,000
	N	2243	2243	2243

**. Correlation is significant at the 0.01 level (1-tailed).

Table 41: Correlations of Sum, NCSS and SumNCSS with Instability

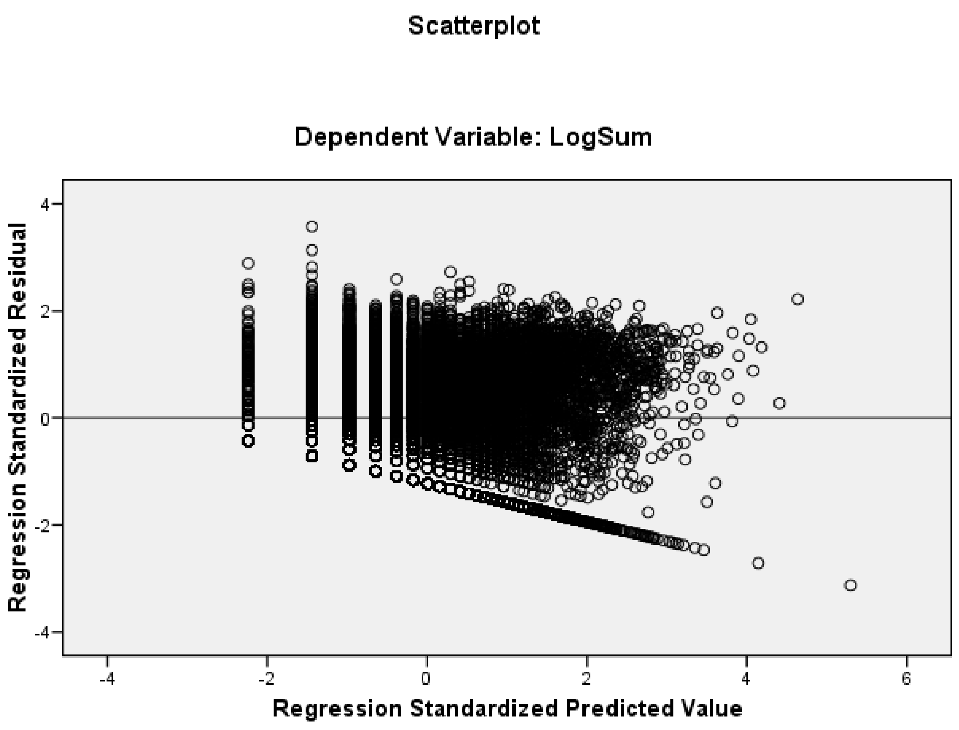

Figure 28: Scatterplot for Residuals and Predicted values of Sum and NOM

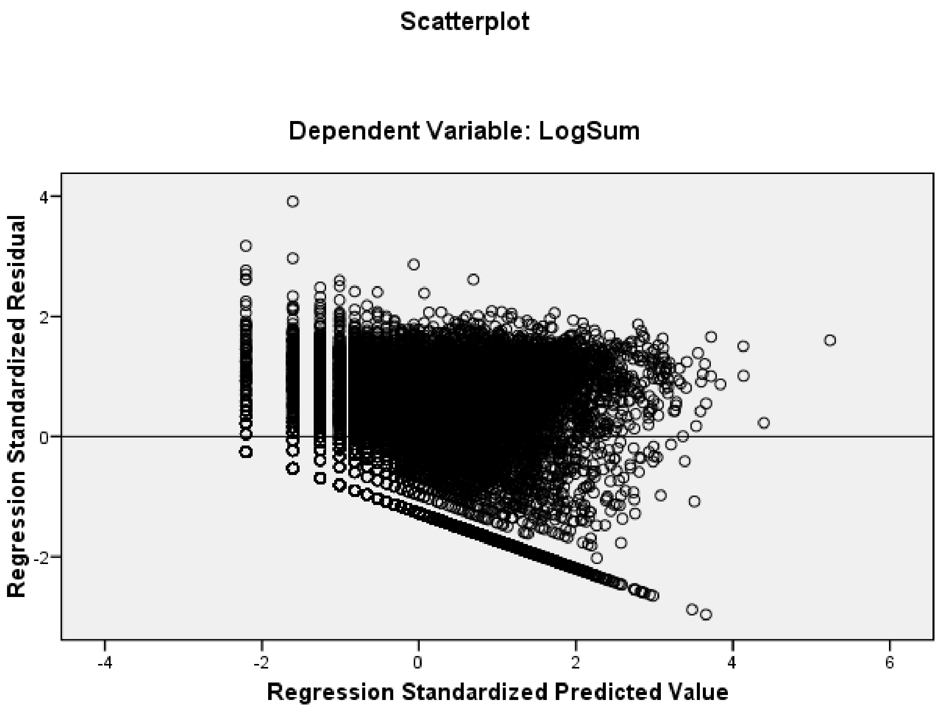

Figure 29: Scatterplot of Residuals and Predicted Values of Sum and WMC

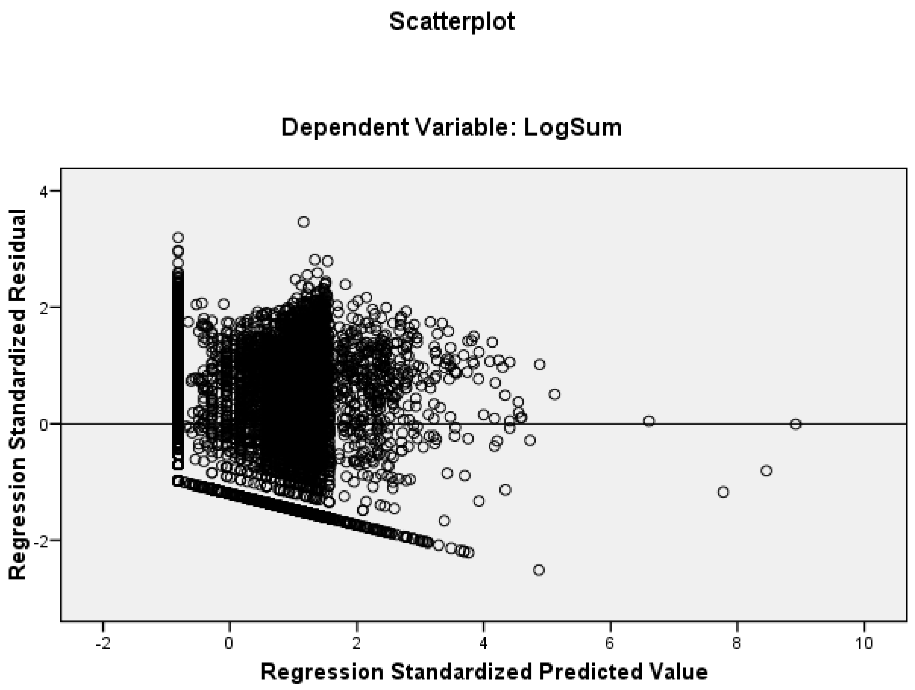

Figure 30: Scatterplot of Residuals and Predicted Values of Sum and LCOM

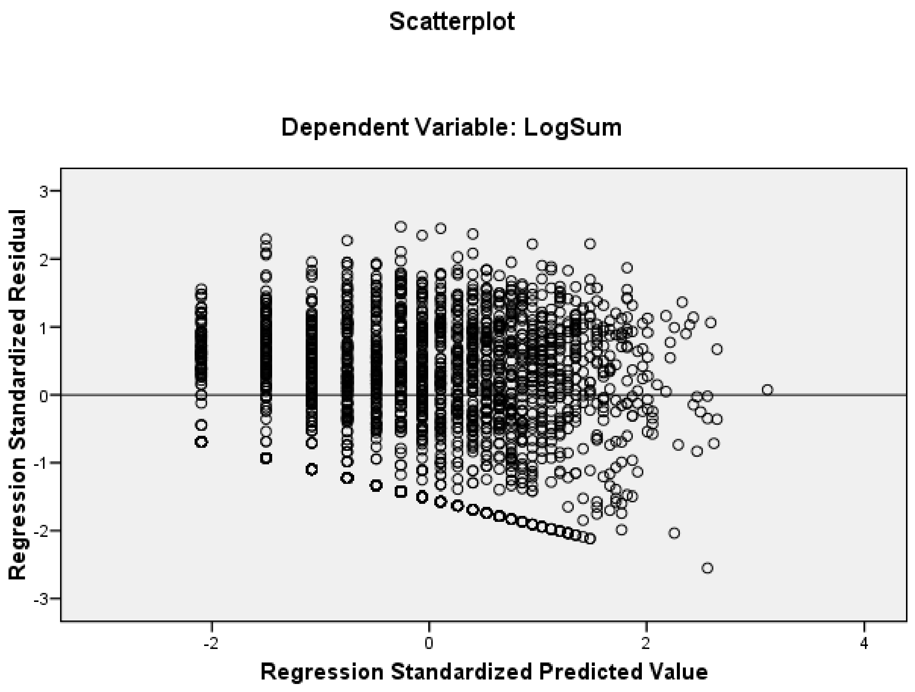

Figure 31: Scatterplot of Residuals and Predicted Values of Sum and Efferent Coupling

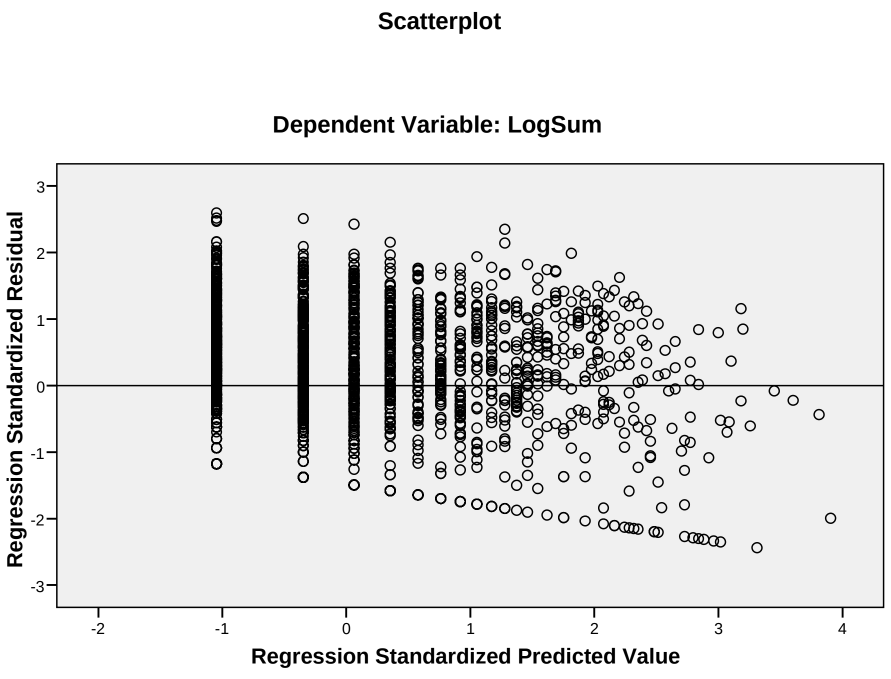

Figure 32: Scatterplot of Residuals and Predicted Values of Sum and Afferent Coupling

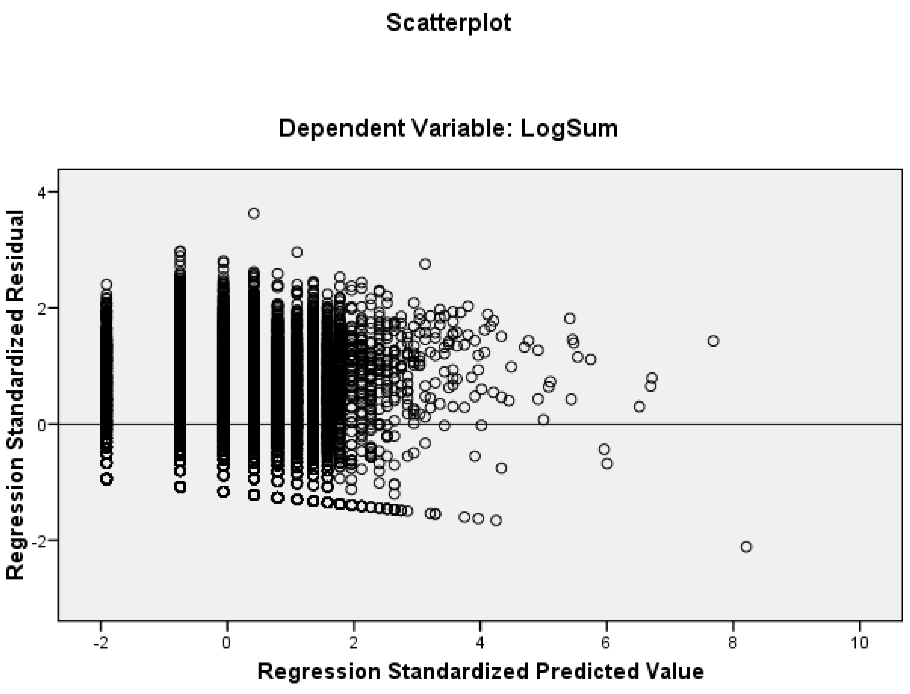

Figure 33: Scatterplot of Residuals and Predicted Values of Sum and DIT

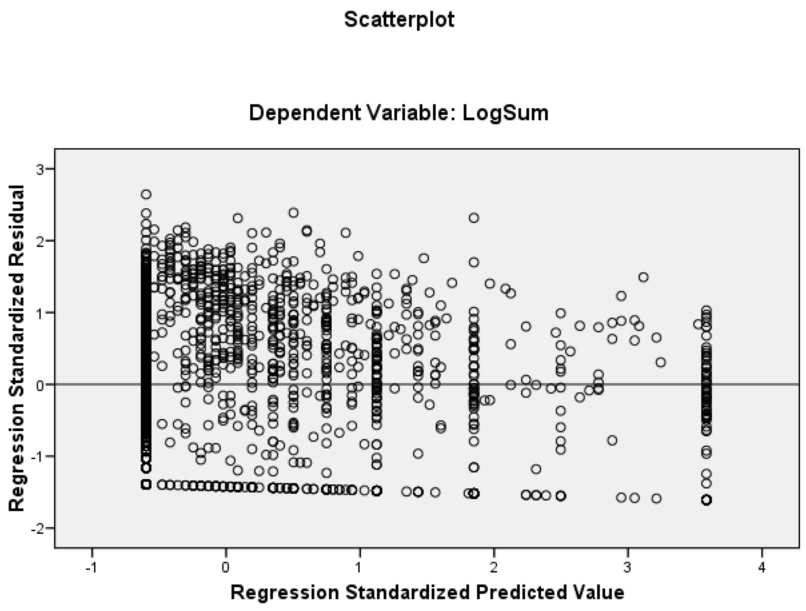

Figure 34: Scatterplot of Residuals and Predicted Values of Sum and Abstractness

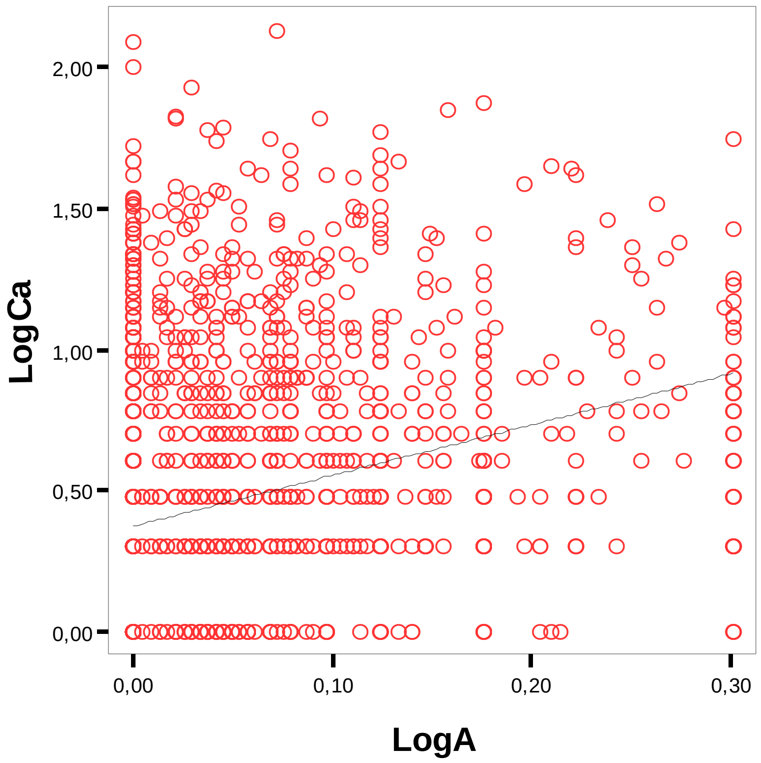

Figure 35: Scatterplot of Ca and A per package

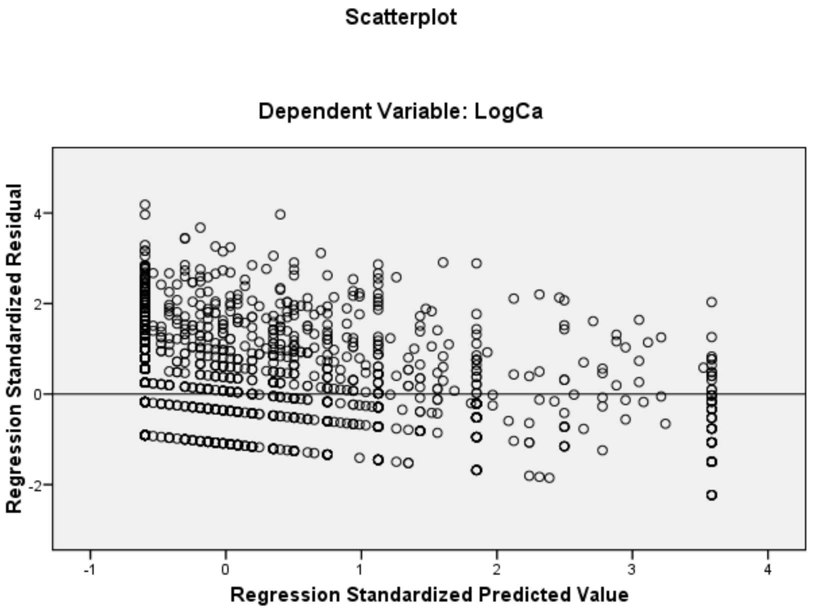

Figure 36: Scatterplot of Residuals and Predicted Values (A and Ca)

Correlations

		LogA
LogCa	Pearson Correlation	,301**
	Sig. (1-tailed)	,000
	N	2243

**. Correlation is significant at the 0.01 level

Table 42: Correlations of A and Ca

Model Summary[b]

Model	R	R Square	Adjusted R Square	Std. Error of the Estimate
1	,301[a]	,091	,090	,41031

a. Predictors: (Constant), LogA

b. Dependent Variable: LogCa

Table 43: Regression of A and Ca