Software Metrics as Benchmarks for Source Code Quality of Software Systems

Julien Rentrop
August 31, 2006

One Year Master Course Software Engineering
Thesis Supervisor: Dr. Jurgen Vinju
Internship Supervisor: Drs. Patrick Duin
Company or Institute: Software Improvement Group
Availability: Public

Universiteit van Amsterdam,
Hogeschool van Amsterdam,
Vrije Universiteit

Contents

SUMMARY 4

PREFACE 5

1 INTRODUCTION 6

1.1 Background 6

1.2 Definitions 6

1.3 Research questions and outline 7

1.4 Disclaimer 7

2 BACKGROUND AND CONTEXT 8

3 SELECTING SOFTWARE METRICS 10

3.1 Selection criteria 10

3.2 Evaluation of software metrics 10

3.3 Conclusion 14

4 BENCHMARKS 15

4.1 Representing software metrics in benchmarks 15

4.2 Validation approach 17

4.3 Conclusion 19

5 BENCHMARK DATA COLLECTOR 20

5.1 Functionality 20

5.2 Design 20

5.3 Summary 23

6 CASE STUDY 24

6.1 Participants 24

6.2 Judgments by assessors 24

6.3 Benchmarks 25

7 CONCLUSION 31

APPENDIX A: MEASUREMENT GUIDELINES 32

BIBLIOGRAPHY 33

Summary

Preface

This project has been conducted at the Software Improvement Group, to whom I am grateful for providing me an interesting environment to work in. I would like to thank my coaches drs. Patrick Duin and dr. Jurgen Vinju for their suggestions and motivation during this internship. Furthermore I would like to thank dr. Harro Stokman and dr. Tobias Kuipers for their support and valuable ideas.

For reviewing earlier versions of this thesis I would like to thank Tim Prijn.

Finally I would like to thank my parents and my brother Michel for their support.

1 Introduction

1.1 Background

The Software Improvement Group (SIG) is a company that is specialized in the analysis of large software systems. Based on these analyses a range of services and products are offered to their clients. An example is the automatic generation of documentation for legacy systems. The generated documentation helps software developers of the client in understanding the source code of the legacy systems. Another example is performing software risk assessments. A software risk assessment assesses the technical quality of a software system. It is based on measurements of the source code and on interviews of the stakeholders involved in the project. The results of these measurements are interpreted by the experts of the SIG, which leads to the identification of problems and possible solutions to overcome the identified problems. These are reported to their clients.

There are multiple questions that a client can have when they ask the SIG to perform an assessment. The question might be as broad as wanting to know whether the software system is maintainable in the future or as specific as wanting to know whether it is easy to implement a change in the length of bank numbers in their system. A question often asked by clients is: is the quality of my software system comparable to other software systems? Knowing the answer of this question helps the client to decide whether they should improve their current software development practices or sustain their current practices.

1.2 Definitions

The term software metrics is used as a collective term to describe a wide range of activities concerning measurements in software engineering [Fenton 99]. In this thesis the use of it is restricted to the following classic definition:

Definition: A software metric measures properties of source code.

With this definition we exclude metrics that can be gathered in a software project that are not based on source code.
The goal of this study is to use metrics to compare the quality of a software system against the quality of a set of other software systems. We use benchmarking as a method for comparison. A benchmark is defined as follows:

Definition: A benchmark is a comparison of an organization's or product's performance against its peers.

In this study the entity that is benchmarked are software systems. To be more precise the source code of software systems. The term performance is a generic term that is used as an indicator of goodness [Sim 03]. The advantage of benchmarking is that it helps organizations to overcome blindness towards other approaches that are more fit then currently employed.

1.3 Research questions and outline

The main research question is defined as follows:
Is benchmarking based on software metrics a good method to determine the maintainability of the source code of software systems?

Software risk assessments, such as performed by the SIG, evaluate software systems that are in or will enter the maintenance phase. The major goal of these assessments is to determine whether the system is understandable and therefore able to keep its business value in a changing environment. This study's proposed benchmarks for source code quality can be used as a tool that helps assessors in determining the maintainability of a system.

This main question is divided in several sub questions:

- Which software metrics can indicate the quality of a software system's source code?
  To answer this question we will investigate software metrics that are introduced in literature. Selection criteria are defined to evaluate the metrics. The results are described in chapter 3.

- How to represent software metrics in a benchmark?
  There are different ways to represent benchmarks. In chapter 4 we define a set of representations and discuss the advantages and disadvantages of each.

- How to design a tool that gathers measurement data to create the benchmarks?
  The SIG has developed a toolset to automatically measure the source code for different programming languages. A contribution of this study is the development of the Benchmark Data Collector tool that extends SIG's toolset to enable benchmarking.

- How to validate the results of the proposed benchmarks?
  In chapter 4 we investigate methods for the validation in literature and propose a new method that can be used to validate the proposed benchmarks in this study. In chapter 6 we apply and validate the proposed benchmarks to 11 industrial used software systems written in the Java programming language.

In chapter 7 the answer to the main research question is given.

1.4 Disclaimer

This document contains software measurement values of real life software systems. As part of a confidentiality agreement the names of these software systems and the companies who created them are altered to make them anonymous.

2 Background and Context

Measurement is needed because it helps in understanding a particular entity or attribute. Even when it's not clear how to measure a certain attribute, the act of proposing and discussing about it helps in creating a better understanding [Fenton 97]. The promise of software metrics is to help managers and engineers in decision making based on facts.

The topic of software metrics can be divided in two parts:

- Product metrics: Measures that quantify attributes of a software system (the product). Examples of attributes that can be measured are the size, complexity and the amount of reuse.
- Process metrics: Measures of the process of creating a software product. Examples of attributes that can be measured are the amount of time spent, defects found and the stability of the requirements.

Current literature of benchmarking is mainly focused on process metrics. One of the main contributors to benchmarking software processes is Caspers Jones. His book includes benchmark studies on the use of best practices, productivity rates and defect rates within different organization types [Jones 00].

This study is focused on benchmarking the maintainability of the source code of software systems. The metrics used are derived from the source code. Many software metrics have been proposed that can measure properties of the source code. A comprehensive overview of these metrics is provided by the Software Engineering Institute [SEI].

Most of the literature about software metrics on the source code is within one system. In contrast, in this study metrics are used to compare at the system level. No literature was found that used the benchmark terminology to compare software systems on source code metrics. However, we did find some studies that are closely related:

- The NASA has built a repository of various metrics for both procedural and object oriented programming languages [NASA-1]. Another repository of metrics created by the NASA also includes metrics about errors and requirements [NASA-2]. Researchers of the NASA disseminated many of their results based on their studies on software metrics. However, they did not provide insight into how the repository of metrics for different software systems can be compared against each other.
- A study in which open source projects are compared against each other and one originally closed source system against its open source successor [Samoladas 04]. The Maintainability Index [Coleman 94] metric was measured in time (for each successive version) to detect whether the system's source code quality is improving or deteriorating.
- Literature that proposes a new method or tool, often also contains a section in which the proposed tool is applied to a set of systems. An example is the proposal of a tool for detecting duplicated code independent of the programming language [Ducasse 99]. In this work the tool is applied to a set of software systems and the results are informally compared against each other.

We expect there are several reasons why no benchmark using source code metrics have been proposed in literature yet. First, it can be hard to measure quality of the source code. Second, it can be hard to obtain the source code of software systems to analyze.

3 Selecting software metrics

In this chapter the answer to the question "Which software metrics can indicate the quality of a software system's source code?" will be given.

To answer this question we will investigate software metrics that are introduced in literature. In the first section we define selection criteria to enable a selection. In the second section describe the individual software metrics and determine whether they match the selection criteria. This chapter concludes with answering the question mentioned above.

3.1 Selection criteria

The following selection criteria have been defined in cooperation with the SIG:

- The software metric must be easily explainable to clients. The benchmarks that are based on the selected software metrics will be presented to clients of the SIG. The clients do not have in-depth knowledge of software metrics. It's therefore necessary that the rationale behind the metrics can be easily explained.
- The software metric must be applicable to many programming languages. Our intent is to enable the creation of benchmarks for different programming languages. We first want to explore general metrics that are applicable to all programming languages instead of metrics that are limited to only a limited set of programming languages. We do however explicitly not demand that a metric in programming language X should be comparable to that same metric in programming language Y.
- The software metric must be automatically calculable from the source code. It must not be necessary to compile and execute the code, because this might not be practically possible because not all external dependencies are available so it's not possible to create an environment to run and measure it in.
- The software metric must have a strong basis in literature. A strong basis in literature will ensure that the applicability of the software metric is known.

3.2 Evaluation of software metrics

3.2.1 Lines of Code

Since the start of software engineering engineers have been counting the lines of code they wrote. Counting lines is used for estimating the amount of maintenance required and it can be used to normalize other software metrics [Rosenberg 97].

In the early days when assembly programming languages were used the notion of a line of code was simple. However when third generation programming languages were introduced the notion of a line of code became harder. Third generation programming languages have programming constructs for structured control flow. For example the begin ('{') and end ('}') tokens of blocks in the C programming language. To standardize the counting the Software Engineering Institute has published a set of recommendations [Park 92]. Broadly two different counting methods exist: Physical Lines of Code and Logical Lines of Code.

3.2.1.1 Evaluation

TODO

3.2.2 McCabe's Cyclomatic Complexity

In 1976 McCabe defined the cyclomatic complexity number metric. The metric measures the number of independent paths through a software module [McCabe 76].

[McCabe 76] proposes an upper limit of 10 for cyclomatic complexity because higher values would indicate less manageable and testable modules. This upper limit is not based on empirical research, however multiple real world projects confirm that modules with higher cyclomatic complexity values often have more errors and are less understandable [McCabe 89].

Although cyclomatic complexity is widely used, critique on it exists. [Shepperd 88] claimed that it's based on poor theoretical foundations and an inadequate model of software development. He also claimed that cyclomatic complexity is a proxy for, and often outperformed by, lines of code.

A reason for the wide use of the cyclomatic complexity metric is that it can be easily computed by static analysis and it's already widely used in the industry for quality control purposes [Fenton 99].

3.2.2.1 Evaluation

The cyclomatic complexity metric is one of the oldest metrics and is still in use in research and practice today. There are however, as described earlier, mixed opinions about the metric. Explaining this metric to clients entails explaining the concept of control flow. With a few examples it's possible to explain this metric to clients. The software metric can be calculated for both procedural and object oriented programming languages. Tools are available to determine the cyclomatic complexity. The cyclomatic complexity has been selected to be a part of the benchmarks.

3.2.3 Object Oriented metrics

The OO approach models the world in terms of objects. This extends procedural languages that are based on data fields and procedures. Traditional metrics such as cyclomatic complexity cannot measure OO concepts such as classes, inheritance and message passing [Chidamber 92].

New metrics have been developed to measure OO systems. One commonly used set of OO metrics is Chidamber and Kemerer's suite of class level metrics:

- Weighted Methods Per Class (WMC)
  WMC is the sum of the static complexity of the methods. When all static complexities are considered to be unity then WMC can be defined as the number of methods.

  The larger the number of methods the greater the impact on sub classes. Classes with many methods can be more application specific and therefore harder to reuse.

- Depth of Inheritance Tree (DIT)
  When a class is deeply nested it inherits more from it's ancestors. This can increase the complexity of the class.

- Number of Children (NOC)
  Classes that have many children are hard to change because of the tight couplings with its children.

- Coupling Between Objects (CBO)
  A high number of couplings with other classes is disadvantageous because when the interface of a class it is coupled to changes it needs to be modified as well.

- Response For a Class (RFC)
  RFC is a measure of the interaction of a class with other classes.

- Lack of Cohesion in Methods (LCOM)
  This metric calculates the usage of a class's attributes in its methods. A class lacks cohesiveness when methods do not make use of its attributes.

An empirical investigation in an academic environment reported five out of six of these metrics to be a useful predictor to class fault-proneness [Basili 96].

An extensive report of the Chidamber and Kemerer's metrics suite and other OO metrics can be found in [Archer 95].

3.2.3.1 Evaluation

The object oriented metrics proposed by Chidamber and Kemerer are often referred to in literature. The metrics can be useful quality indicators, but there are some limitations. For example Coupling Between Objects counts couplings to classes in the same package, different packages and packages of external libraries all the same. Couplings with a lot of external packages are much worse then couplings to internal packages.

The metrics are designed for the OO paradigm. The goal of this project is to use metrics to create benchmarks for a wide number of different programming languages. The OO metrics are therefore not selected.

3.2.4 Duplicated code

When code is duplicated it can become harder to make changes because one change must also be made in all copies. This takes more time and is also error prone as it's easy to forget to make the change in multiple places.

One way to measure duplicated code is by performing line based text matching [Baker 95]. Another way to measure duplicated code is by matching layout, expression and control flow metrics [Mayrand 96].

A software system can have unique source code in multiple places that provide the same functionality. This is called conceptually duplicated code. The programmers of systems with conceptually duplicated code have programmed the same functionality multiple times and did not reuse or copy existing code. Conceptually duplicated code cannot be detected automatically and is therefore detected by inspections performed by humans.

3.2.4.1 Evaluation

Duplicated code is a topic that has got a lot of attention in scientific literature. The concept of duplicated code is simple: A lot of duplicated code unnecessary increases maintenance costs because the system is larger. This concept is easily explainable to clients. Duplicated code can be determined for all programming languages. There are

even tools available (such as [Kettelerij 05]) that can detect duplicated code independent of the programming language.

3.2.5 Dead code

Dead code is code that is never executed. Having dead code increases the amount of code that needs to be maintained. The programmers that maintain the source code might not know whether the code is dead or still used. Dead code makes it harder to maintain the system's source code.

A reason for dead code to remain in the source code is that programmers are afraid that the system might break without it. A comprehensive unit test suite can give more assurance that the system doesn't break without the dead code. Another reason for keeping dead code is that programmers want to be able to restore old code. Keeping old code should not be necessary because version control systems can easily restore old code.

3.2.5.1 Evaluation

The percentage of dead code in a software system is a useful indicator of the quality of the source code. A problem however is that developing a tool that can automatically determine which parts of the source code is dead is far from easy. One of the problems is that dynamic constructs in a programming language are hard to resolve.

Another problem is that all entry points must be available. In practice the SIG does not always have access to the source code of external systems. These external systems can be entry points. If not all entry points are available source code can be detected as dead when it isn't.

3.2.6 Database metrics

A lot of research has been focused on the measurement of source code of programs. However many information systems make extensive use of databases and therefore measuring the quality of the database structure is important as well. To measure the maintainability of a database three simple metrics have been proposed [Calero 01]:

- Number of tables
- Number of columns
- Number of foreign keys

3.2.6.1 Evaluation

Software systems often make use of databases. A benchmark that can indicate the quality of the database structure would therefore be desirable. Research on the definition of software metrics for databases is however still in its infancy. The metrics proposed by [Calero 01] can be useful to compare and select from different database models designed for one problem domain but are not useful as indicators for quality to compare database schemas designed for different problem domains. The metrics can be used as an absolute size indicator such as total lines of code for source code.

3.3 Conclusion

In this chapter we defined selection criteria and applied them to a number of different metrics. Based on the selection we found the lines of code, cyclomatic complexity and code duplication metrics useful quality indicators of a software system’s source code.

4 Benchmarks

This chapter consists of two parts. The first section will answer the question "How to represent software metrics in a benchmark?". To validate the proposed representations, the second section will answer the question "How to validate the results of the proposed benchmarks?".

4.1 Representing software metrics in benchmarks

In this paragraph we propose three different ways to represent a benchmark using the metrics selected in the previous chapter.

4.1.1 Ordered tables

Some of the metrics discussed in the previous chapter measure properties at the module level. The module level means classes or methods in Object Oriented languages and units and procedures in procedural programming languages. A concrete example is McCabe's cyclomatic complexity that is calculated per method in Java.

In the benchmarks proposed here, systems are compared and not modules within one system. To use these module level metrics we need a way to lift the values of the module level to the system level. This lifting is called aggregating. There are a number of different aggregation functions such as average, median, sum, max and min. The sum is not useful because the size of an application would affect the measurement results. The max and min function would make the result based on only one module. In this research we use the average aggregation function to lift metric values to the system level.

As defined in the introduction a benchmark is a comparison of an organization's or product's performance against its peers. In the benchmark presented here we use individual metrics as performance (quality) indicators and the comparison is presented by sorting the metric values. A system that has a high position in the table means that it's the best in the benchmark and vice versa. A dummy example is presented below with four different systems named A, B, C and D:

System name	Metric value
D	2
A	8
C	9
B	14

In this (dummy) benchmark system D scores best, A and C are in the middle and B is worst.

Now we use the software metrics selected in the previous chapter, define the level of measurement, choose an aggregation function and determine how the metric results should be ordered:

Benchmark	Metric	Level	Aggregation function	Order
B1	Lines of code	Method	Average	Ascending
B2	Lines of code	Class	Average	Ascending
B3	Cyclomatic complexity	Method	Average	Ascending
B4	Cyclomatic complexity	Method	Percentage of methods below threshold *	Ascending
B5	Cyclomatic complexity, Lines of code	Method	Percentage of code below threshold *	Ascending
B6	Maintainability Index**	System	None	Descending
B7	Code duplication	System	Percentage of duplicated code	Ascending

* In [McCabe 76] the threshold value 10 is proposed to indicate modules that are of low quality
** Maintainability index is a compound metric that is based on the averages of lines of code, cyclomatic complexity, Halstead volume and comment percentage [Coleman 94].

We expect that some of these benchmarks will result in highly similar results:

- Lines of code at the method level (benchmark 1) and lines of code at the class level (benchmark 2) is expected to be identical because large methods would make a class large as well.
- Cyclomatic complexity (benchmark 4) and cyclomatic complexity with lines of code (benchmark 5) are expected to be similar.

These expectations will be investigated in chapter 6.

4.1.2 Combined benchmark

In the previous paragraph 7 different benchmarks are proposed. Multiple benchmarks are useful because they can give an indication of different aspects of quality. A system might for example have highly complex methods but does have a low percentage of duplicated coder. However it would be interesting if the results of the different benchmarks can be combined in one benchmark to get an overall view.

The unit of measurement of each benchmark is different. Simply calculating the mean of all measurements is therefore not valid. To overcome this problem we can calculate the difference between the mean and each system and divide this by the standard deviation. Now we have one unit of measurement for all benchmarks that tells how many standard deviations a measurement is away from the mean. An average can be calculated for any combination of benchmarks. Which benchmarks should be combined will be investigated in chapter 6.

4.1.3 Histograms

In this paragraph we propose to use histograms as a representation that can compare the distribution of lines of code per class. To create the histogram all systems the lines of code per class measurements are partitioned in bin ranges. We use 11 bins each with a width of 50. We use relative frequencies instead of actual

frequencies because the sizes of the systems measured this study vary. In a relative histogram the Y-axis runs from 0% to 100%.

Our approach is to compare one system's distribution against the *benchmark*. The benchmark is the average distribution of all other systems.

A significant difference of the distribution of a software system and the benchmark does not directly tell that it's worse or better then average. So it doesn't exactly fit the definition of a benchmark. However a significant difference is interesting and does deserve attention of software assessors to determine the causes of the difference. In this way using measurement results of other systems that are analyzed helps in analyzing the current system.

To quantify the difference between the benchmark and one system we adopted the Histogram Difference Measure (HDM) from [Cui 06]. HDM is calculated as follows: Given two histograms with the same number and bin widths: One system's histogram and the benchmark histogram. Bin difference is the absolute difference between two bins (the system's and the benchmark's bin). Then histogram difference corresponds to the summation of bin differences. HDM is the normalized histogram difference. HDM has an interval of 0 to 1 where 0 indicates that it's totally different and 1 indicates a perfect match. A perfect match in this case means that the system is equally distributed as the benchmark.

4.2 Validation approach

Software metrics are used for different goals. Some of these goals are for finding mod7ules that are likely to have errors [Menzies 02], finding modules that are hard to test and finding modules that are hard to understand and therefore hard to change [McCabe 76].

4.2.1 Literature approaches

In this paragraph we discuss approaches used in literature to validate the use of software metrics.

4.2.1.1 <u>Interviewing developers</u>

This approach consists of running measurements on one or more systems. From these measurement some values are selected, for example the highest and lowest values. Interviews with developers of the software system are conducted to find if the opinion of the developers confirms the metric values such as in [McCabe 76] and [Chidamber 96].

This approach can be applied to compare metrics at the module level. The developers of a software system can state whether they find one module easier or harder to understand and modify then other modules. However, using interviews with developers to validate comparisons at the system level is harder. Asking developers whether they find the systems they maintain to be easy or hard to modify will not be useful:

- A software system can be huge. A developer might only be an expert of a part of the system. The developer cannot state the quality of the whole system.
- Research showed that a crucial factor in software maintenance is a stable maintenance team [SWEBOK 04]. Developers that have worked a long time on

maintaining a system gain a lot of knowledge about it and will therefore find it easy to modify.
- A software developer has only worked on a limited amount of different systems. A developer's opinion about the quality of the system is affected by the systems the developer had worked on. It's possible that all of them are quite good or bad but in the developer's eyes some can be worse and some are better.

We therefore conclude that interviewing developers is not suited for validating the proposed benchmarks.

4.2.1.2 Correlating with discovered bugs

Another approach is to compare the measurements with the amount of bugs reported such as in [Basili 96] and [Menzies 02].

This approach is applicable for one system or for all systems in one organization, but is far more difficult to do across different organizations. There are multiple reasons for this difficulty:
- Different organizations record (or do not record at all) bug data in different ways that can make comparisons across organizations impossible.
- Bugs can be detected at various stages such as during development, after development and when the system went in production. The duration of these stages can be different for each software system and therefore a comparison would be invalid.

Furthermore the goal of our benchmarks is used to determine whether source code is maintainable and not about bugs/correctness. We therefore conclude that correlating discovered bugs is not suited for validating the proposed benchmarks.

4.2.2 Our validation approach

We need to validate if the proposed benchmarks give an accurate ordering of the software system's source code quality. As validation approaches in literature are based on comparisons within one system we defined a new validation approach that is based on a comparison with judgments given by the assessors.

The judgments of the assessors can be found in assessment documents. Per system assessed one document is written. An assessment document reports on the quality of different subjects of a software system, for example the architecture/design, source code, (unit) testing, documentation and tool usage. In our validation approach we only look at the judgments given about the quality of source code. The other subjects are outside the scope of this study. The judgments in the reports are written in natural language. As human language is ambiguous we rewrite the judgments in natural language to a fixed set of judgments: high quality, normal quality and low quality.

Judgments of assessors of the SIG are based on the interpretation of measurement results. The metrics used by assessors are often the same as the metrics used in the benchmarks: Lines of Code, McCabe and Code duplication. The assessors do have more tools to determine the quality of the source code like focusing metrics on specific parts of a system and manually inspecting the source code.

To give a judgment the assessors implicitly compare the measurements of the current system with measurements of other systems they have assessed. It's

possible that the benchmarks are more accurate then the judgments of assessors as one assessor didn't assess all systems and might not remember values from previous assessments.

4.3 Conclusion

Three different ways to benchmark software systems have been proposed. The first one consists of ordered tables wherein the position of the software system in the table represents its quality against other software systems, the second one consists of using multiple benchmarks to create one combined benchmark and the third one uses histograms which can be visually inspected to find differences in the distribution of source code in software systems.

By studying literature we found two approaches to validate software metrics. We found that these approaches are not applicable to validate the benchmarks proposed in this study. A new validation method has been proposed that is based on judgments of assessors that will be used to validate the proposed benchmarks.

5 Benchmark Data Collector

This chapter describes the goals and design of the benchmark data collector. First, the data that is collected is discussed. Second, we describe the design considerations that are made.

The Software Improvement Group has built the System Analysis Toolkit (SAT) to analyze software systems. Among others it contains the implementation of software metrics such as lines of code, cyclomatic complexity and code duplication. The SAT is written in Java.

5.1 Functionality

The goal of this study is to benchmark software systems by use of metrics on source code. To achieve this goal two components must be available:

- A filled database with metrics of software systems
- A view of this database that shows the benchmark results

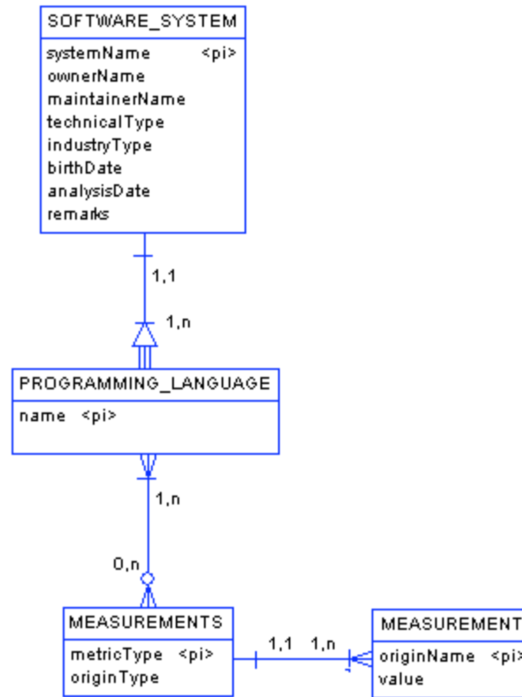
This chapter deals with the first point by developing a tool that collects measurement data and stores it in one database. The second point is achieved by standard tools such as the MySQL console application for running SQL queries and Microsoft Excel for presenting the measurements in tables and charts. The development or selection of a full fledged presentation application is outside the scope of this project.

Currently the Benchmark Data Collector can collect data for software systems programmed in Java and in COBOL. In the future this tool can be extended so it can be used for other programming languages as well.

5.2 Design

5.2.1 Entity Relationship (ER) model

To describe the data that is stored the following E/R data model is created:



E/R Diagram

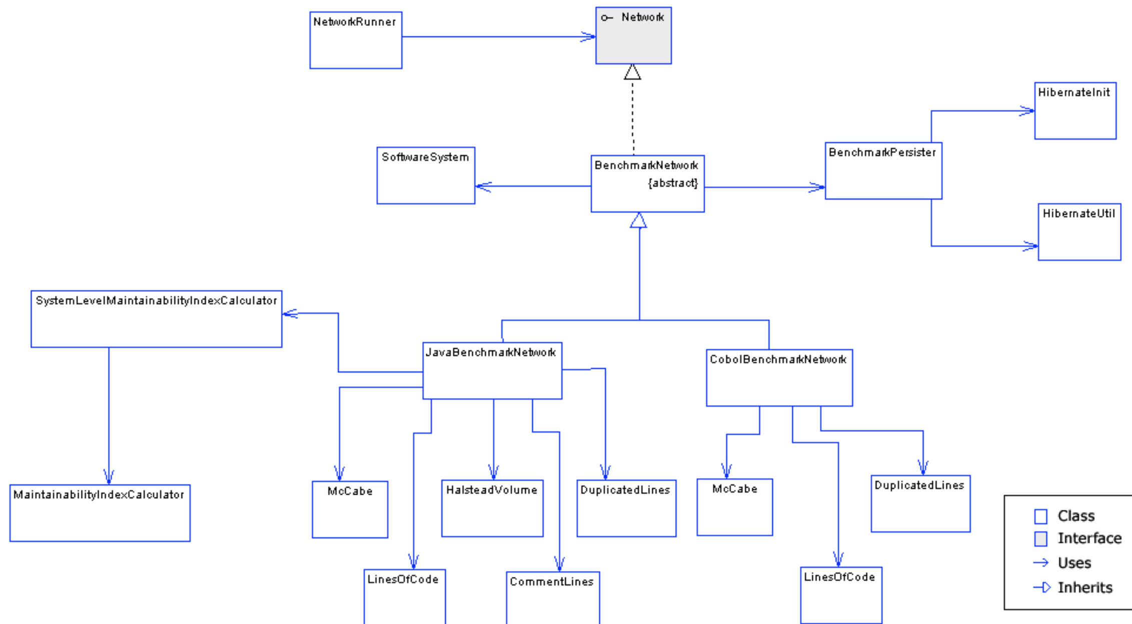
In this diagram four entities are defined. SOFTWARE_SYSTEM contains attributes to identify and categorize a software system. The attribute ownerName defines the company or organization that owns the system. The attribute maintainerName can be the same as the ownerName, but is different in case of outsourcing. Storing these attributes is relevant because it would be interesting to benchmark all systems developed by or for one owner or maintainer against each other. The system is categorized by it's industryType (for example Finance, Public and Industry) and it's technicalType (for example Standalone application, Web application and Mainframe).

A software system can consist of multiple programming languages. In this data model measurement results are separated per programming language. For example one software system is programmed in C and ASM and has a SLOC of 10.000 for C and a SLOC of 5.000 for ASM.

The MEASUREMENTS entity has two attributes: metricType and originType. The metricType attribute is used for storing the name of the metric, for example Lines of Code. The originType attribute stores the level of measurement, for example system level and method level.

5.2.2 Class diagram

The Benchmark Data Collector is written in Java. The following diagram gives an overview of the classes and their relations:



Class diagram of Benchmark Data Collector

5.2.3 Design considerations

During the development we made a number of considerations are made that affects the design. The most important are described below:

5.2.3.1 Re-using components of the SAT

A goal when developing the Benchmark Data Collector was to minimize the amount of new source code that needs to be written. More source code would cost more time to develop and would require more time to maintain. To reduce the amount of new source code that needs to be written we chose to make use of components that are already available or used in SIG's System Analysis Toolkit:

- Software metrics: The software metrics available in the SAT.
- Persistence: The Hibernate object/relational persistence service [Hibernate]. To simplify the interaction with Hibernate we use utility classes of the SAT.
- File filters: To select which files are analyzed and which aren't we use file filters of the Apache's Jakarta Commons project [Jakarta].
- Input format: The input of the Benchmark Data Collector is a Spring configuration file [Spring]. See paragraph 5.2.3.2.

5.2.3.2 Input: Flexibility by configuration or hard-coded

The input of the benchmark data collector is a Spring XML configuration file. Spring is a Java framework that minimizes hard coded dependencies to increase modularization and testability [Spring]. Spring uses the Inversion of Control (also called Dependency Injection) pattern.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
  <bean class="java.lang.String" id="rootDir">
    <constructor-arg><value>/home/julien/sources/A</value></constructor-arg>
  </bean>

  <bean class="org.apache.commons.io.filefilter.SuffixFileFilter" id="fileFilter">
    <constructor-arg><value>.java</value></constructor-arg>
  </bean>

  <bean class="software_improvers.model.benchmark.SoftwareSystem" id="softwareSystem">
    <property name="systemName"><value>System A</value></property>
    <property name="ownerName"><value>Company A</value></property>
    <property name="industryType"><value>Finance</value></property>
  </bean>

  <bean class="nl.sig.network.benchmark.JavaBenchmarkNetwork" id="network">
    <property name="softwareSystem"><ref bean="softwareSystem"/></property>
    <property name="rootDir"><ref bean="rootDir"/></property>
    <property name="fileFilter"><ref bean="fileFilter"/></property>
  </bean>
</beans>
```

A XML configuration file

The XML configuration file of the benchmark data collector is used to define which code should be analyzed and enter information of the software system. A configuration file is made for each system that is analyzed.

A question raised during the development was: Should it be possible to configure metrics in the XML configuration file? We first thought that it would make the tool more flexible because it would be possible to add more metrics and set different counting option. However we found that introducing these configuration options would introduce problems:

- When there are configuration options it will be possible that metric values are calculated different per system. For example for one system the minimum clone size that is detected is configured at 10 lines and for other systems it’s 6 lines. These difference configurations would make comparisons invalid.
- Configuring takes time from the user. Using the Benchmark Data Collector should take as less time as possible.

The Benchmark Data Collector must promote standardization instead of configuration. We have therefore chosen to define which metrics and options are used in Java code using constants so it’s impossible for the user to use different metrics or options.

5.3 Summary

In this chapter we described the design of the Benchmark Data Collector. The Benchmark Data Collector collects measurements results from the source code for software systems written in Java or COBOL.

6 Case study

In this case study we apply and validate the benchmarks defined in chapter 4. The Benchmark Data Collector tool described in chapter 5 is used to gather the measurement values.

6.1 Participants

The benchmark consists of 11 software systems written in Java. The functionality provided, the industry type and sizes of these systems are diverse.

Software System	Description	Size in SLOC
A	An administrative application for a governmental organization.	71.313
B	Application B enables micro payments on internet sites.	79.610
C	Supports the organization in managing debits. The application contains batch jobs to automatically invoke actions, like sending letters.	193.551
D	Administrative application for banking products used by regional offices.	95.987
E	Administrative application for leases of cars	28.334
F	Administrative application for an organization in the transport industry.	1.393.551
G	A back-office application for processing postal items.	102.802
H	An interface application that provides clients information about the status of their ordered products.	45.661
I	Administrative application that manages personal information for a governmental organization.	20.167
J	This application is developed and used by the SIG for the analysis of software systems.	72.743
K	An application built for the insurance industry. The application contains an advice module and supports the registration of information about persons.	201.052

6.2 Judgments by assessors

As described in paragraph 4.2.2 we use the judgments of assessors to compare with the results of the benchmarks. The table below describes the judgments per software system:

As the judgments are given in natural language it's not quantifiable. The judgments in the reports can take up multiple sentences and the terms used vary by report. Examples of judgments are "this system is above average", "better then we have ever seen" and "the system is overly complex". We have simplified these sentences to either "good" denoted as "+" or "bad" denoted as "-". The reports we studied didn't judge any system as being average.

Software system	Judgment
A	+
B	+
C	-
D	-
E	+
F	-
G	?
H	?
I	+
J	+
K	-

Two systems (G and H) have not been assessed (yet). The judgment of these systems is undefined. It's not possible to validate these system's places in the benchmark. We did however choose to keep them in, as it will affect the comparison of other systems.

6.3 Benchmarks

In appendix A we describe which parts of the software system are measured. First we present the results and afterwards we compare these results with the judgments given by assessors.

The benchmarks measured here are described in paragraph 4.1.

6.3.1 Ordered table

B1	Value
E	4.77
A	5.28
J	5.67
K	5.76
H	6.15
I	6.24
B	7.94
G	8.65
F	9.00
D	9.56
C	13.55

B3	Value
J	54.25
K	62.40
E	67.62
B	81.32
I	82.31
A	93.34
G	96.89
D	107.01
C	182.08
F	190.22
H	206.61

B5	Value
J	99.16%
K	95.98%
E	93.95%
F	86.60%
H	86.29%
A	84.69%
B	83.73%
G	83.69%
I	78.55%
C	76.33%
D	52.49%

B7	Value
J	3%
E	3%
I	5%
B	5%
G	9%
K	10%
F	16%
H	18%
A	18%
D	20%
C	29%

B2	Value
E	4.77
A	5.28
J	5.67
K	5.76
H	6.15
I	6.24
B	7.94
G	8.65
F	9.00
D	9.56
C	13.55

B4	Value
J	99.89%
K	99.51%
E	99.33%
H	99.03%
A	98.69%
F	97.98%
G	97.96%
B	97.72%
I	97.37%
C	93.99%
D	91.56%

B6	Value
E	161.88
K	159.64
A	158.14
I	153.25
B	149.19
G	147.34
D	146.86
F	145.36
H	143.85
J	139.18
C	125.71

6.3.1.1 Validation

A limitation of the validation described here is that the number of systems analyzed here is rather small (11 software systems). It's possible that with more systems the results would be different.

The following table presents the judgments given by the assessors and the position in each benchmark. To compare the position a software system to a judgment we have simplified the positions in the benchmark to:
+ = High (Top four systems)
0 = Average (The three systems in the middle)
- = Low (Bottom four systems)

The systems denoted as "0" in the benchmarks are not comparable to the judgments because no system is judged as average. We have chosen to exclude these software systems because if only one or two systems are added the quality can jump from average to low or to high quality.

To make the comparison easier the judgments of the assessors are repeated in the second column. When a system's position in the benchmark confirms the judgment it's colored green and it's colored red when it's contradicts the judgment.

Software System	Judgment	B1	B2	B3	B4	B5	B6	B7
A	+	+	0	+	0	0	+	-
B	+	0	+	-	-	0	0	+
C	-	-	-	-	-	-	-	-
D	-	-	-	-	-	-	0	-
E	+	+	+	+	+	+	+	+
F	-	-	-	-	0	+	-	0
G	?	-	0	0	0	-	0	0
H	?	0	-	0	+	0	+	-
I	+	0	0	0	-	-	+	+
J	+	+	+	+	+	+	-	+
K	-	+	+	+	+	+	+	0

When we ignore the systems that are average (denoted as 0) there are the following differences:

- Benchmark 1: System K
- Benchmark 2: System K
- Benchmark 3: System B, system K
- Benchmark 4: System B, system I, system K
- Benchmark 5: System F, system I, system K
- Benchmark 6: System J, system K
- Benchmark 7: System A

Six out of seven benchmarks wrongly indicate system K as high quality. According to the assessors this system was over engineered. An over engineered software system can have many small and simple methods and classes and therefore score well in the benchmarks 1 to 6. The absolute size and design complexity is considered to be too large for this system

The developers of this system used a too named Checkstyle that automatically reports on overly large and complex classes and methods [Checkstyle].

Another interesting result is that system J scores well on all benchmarks except on the Maintainability Index (B6). We found that this system has a poor result because it had the lowest comment percentage (4.5%). We believe that the comment percentage metric is questionable because code that is easily readable should not need lots of comments to clarify it [McConnell 04].

Based on these results benchmark 1, 2 (average lines of code of methods / classes) and 7 (code duplication percentage) confirm the judgments of assessors.

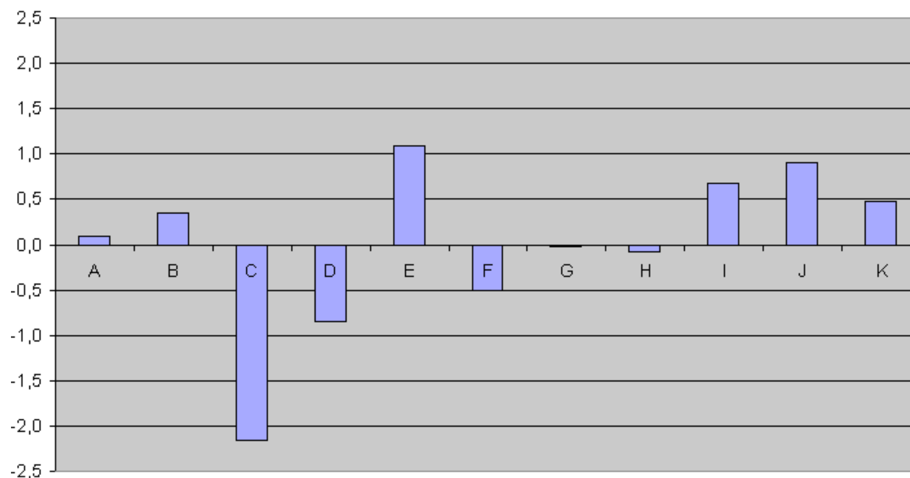
A problem with comparing the judgments to the benchmarks is that one benchmark only measures one aspect of quality while the judgments given by the assessors measure all aspects. Therefore the differences and similarities found can be cause because other aspects were of a certain quality which happens to be the same as the benchmark. As the scales of comparison is only "+" and "-" there is always 50% chance that the results are similar.

The validation would have been more precise if there would be a direct mapping to the aspect measured in the benchmark and the judgment of the assessors. Another improvement would be if the scales were more precise so that the chance of coincidental similarities would be lower.

Next to these problems of the validation method it would be desirable to have a larger set of software systems. A problem is that the benchmarks proposed here are intended to be used by the SIG to help assessors in deciding whether the system is good or not. Gathering more data from software systems would mean this work should not be used because it would influence the assessors. By doing this we and up with a classic chicken and egg problem because for this validation approach we need the judgments of assessors and the assessors need benchmarks for making judgments.

6.3.2 Combined benchmark

In the previous paragraph we found benchmark 1, 2 and 7 to be closest to the judgments of assessors. Now we combine two of these, benchmark 1 (lines of code per method) and benchmark 7 (code duplication), to create one combined benchmark. Both benchmarks are equally weighted in the equation. A value above zero indicates above average quality and vice versa.



Except from system K all values are similar to the judgments of the assessors.

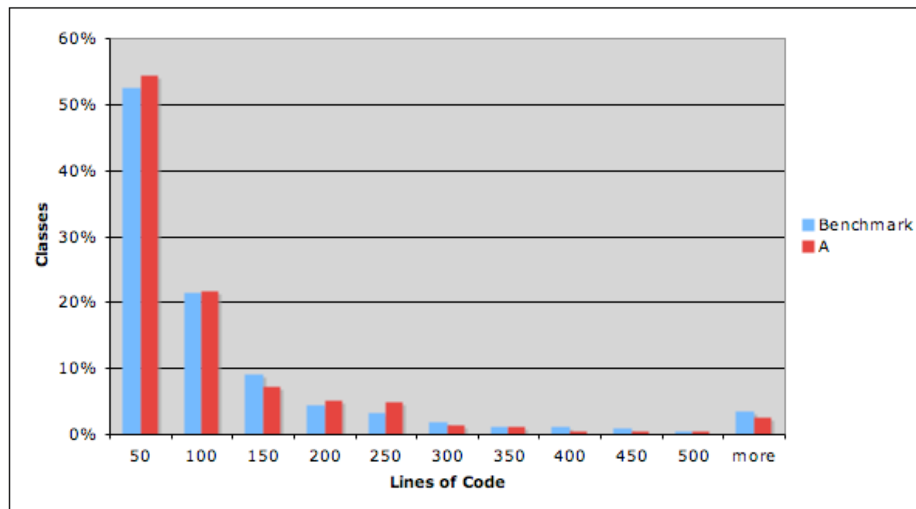
6.3.3 Histograms

The following table presents the differences between the distribution of lines code for each system and the benchmark:

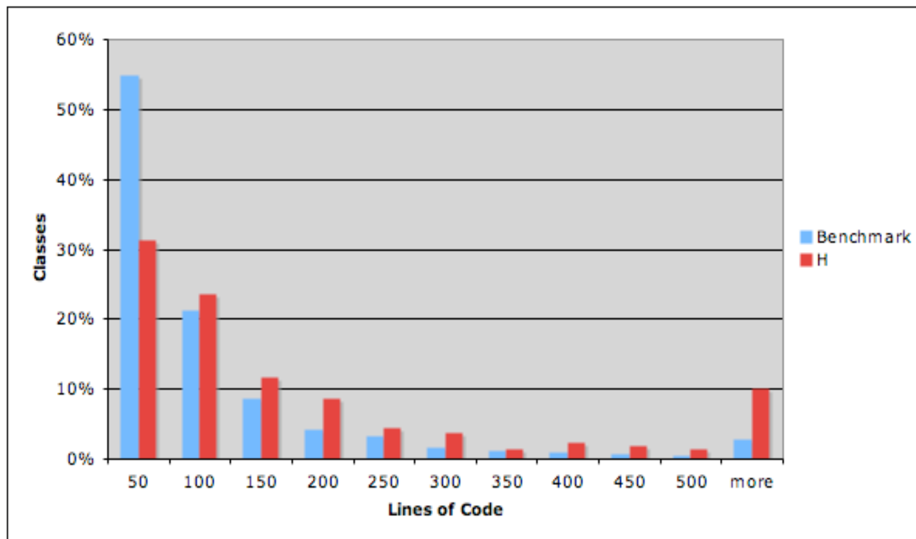
System	HDM
A	0.95773
B	0.94979
C	0.87753
D	0.81657
E	0.92293
F	0.81493
G	0.93672
H	0.76287
I	0.88198
J	0.84791
K	0.90014

Difference between the distribution of lines of code over classes.

We took two entries from this table: System A because its highest value indicates a high similarity with the benchmark and system H because its lowest value indicates that it's most different from the benchmark.



The lines of code distribution of system A is equivalent to the benchmark.



There is a significant difference between system H's and the benchmark's distribution. System H has fewer classes in the smallest bin range and therefore more classes in the higher bin ranges.

The histograms clearly show that system A is equivalent to the benchmark and that system H is different. System H has fewer small classes and has a higher percent of its classes in the more bin. System H's large classes are worth investigating as these classes might take up too much functionality.

7 Conclusion

In this study we explored the use of metrics as benchmarks for the source code quality of software systems. Based on the selection criteria, described in chapter 3, the lines of code, cyclomatic complexity and code duplication metrics were selected. These metrics were used to create representations of the benchmarks.

Is benchmarking based on software metrics a good method to determine the source code quality of software systems?

Appendix A: Measurement Guidelines

To be able to make a fair comparison it's needed to set clear guidelines about what should be measured or not.

Parts to measure

Source code that needs to be measured is all program source code that is maintained by the developers of the project. The following guidelines are defined:

- Program source code is included.
- Source code of external libraries is excluded. External means libraries that are from a third party, for example COTS or open source.
- Source code of libraries made by the company it self should be included.
- Generated code is excluded. However if the generated code is maintained by hand it should be included.
- Unit test code should be excluded. The purpose of unit test code is different then for program code. Unit test code often has less control flow statements and can seriously affect the measurement results. Measuring qualities of unit test code should be measured separate from program code. Measuring unit tests is beyond the scope of this project.

Bibliography

[Jones 00] C. Jones*. Software Assessments, Benchmarks and Best Practices*, Addison-Wesley, 2000

[McConnell 04] S. McConnell. *Code Complete Second Edition*, Microsoft Press, 2004

[Mens 02] M. Mens, S. Demeyer. *Future Trends in Software Evolution Metrics*, ACM, 2002

[Baker 95] B.S. Baker. *On Finding Duplication and Near-Duplication in Large Software Systems*, Proceedings of the Second Working Conference on Reverse Engineering (WCRE '95), 1995

[Mayrand 96] J. Mayrand, C. Leblanc, E.M. Merlo. *Experiment on the Automatic Detection of Function Clones in Software System Using Metrics*, International Conference on Software Maintenance (ICSM '96), 1996

[Veerman 03] N. Veerman. *Revitalizing Modifiability of Legacy Assets*, Proceedings of the Seventh European Conference on Software Maintenance And Reengineering (CSMR'03), 2003

[Brand 97] M.G.J. van den Brand, P. Klint, C. Verhoef. *Re-engineering needs Generic Programming Language Technology*, ACM, 1997

[McCabe 76] T.J. McCabe, *A Complexity Measure*, Proceedings of the 2nd international conference on Software engineering, 1976

[McCabe 89] T.J. McCabe, C.W. Butler. *Design Complexity Measurement and Testing*, Communications of the ACM, 1989

[Shepperd 88] M. Shepperd. *A critique of cyclomatic complexity as a software metric*, Software Engineering Journal, 1988

[Chidamber 91] S.R. Chidamber, C.F. Kemerer. *Towards a metrics suite for object oriented design*, International workshop on Principles of software evolution, 1991

[Chidamber 94] S.R. Chidamber, C.F. Kemerer. *A Metrics Suite for Object Oriented Design*, IEEE, 1994

[Basili 96] V.R. Basili, W.L. Melo. *A Validation of Object-Oriented Design Metrics as Quality Indicators*, IEEE Transactions on Software Engineering, 1996

[NASA-1] NASA Software Assurance Technology Center, Software Metrics Research and Development. http://satc.gsfc.nasa.gov/metrics/. Last visited august 2006

[NASA-2] NASA Independent Verification and Validation Facility, Metrics Data Program, http://mdp.ivv.nasa.gov/. Last visited august 2006

[Rosenberg 97] J. Rosenberg, *Some Misconceptions About Lines of Code*, metrics, p. 137, Fourth International Software Metrics Symposium (METRICS'97), 1997

[Park 92] R.E. Park, *Software Size Measurement: A Framework for Counting Source Statements*, Software Engineering Institute (CMU/SEI-92-TR-020), 1992

[Coleman 94] D. Coleman, D. Ash, B. Lowther, P. Oman, *Using Metrics to Evaluate Software System Maintainability*, *Computer*, vol. 27, no. 8, pp. 44-49, Aug., 1994

[Archer 95] C. Archer*. Measuring Object-Oriented Software Product*, Software Engineering Institute (SEI-CM-28), 1995

[Tian 95] J. Tian, M.V. Zelkowitz, *Complexity Measure Evaluation and Selection*, IEEE Transactions on Software Engineering, vol. 21, no. 8, pp. 641-650, Aug., 1995.

[SEI] Carnegy Mellon Software Engineering Institute: A taxonomy of quality measures. http://www.sei.cmu.edu/str/taxonomies/.

[Calero 01] C. Calero, M. Piattini, M. Genero. A Case Study with Relational Database Metrics, ACS/IEEE International Conference on Computer Systems and Applications (AICCSA'01), 2001

[Demeyer 99] S. Demeyer, S. Ducasse. *Metrics, Do They Really Help?*, LMO, 1999

[Demeyer 01] S. Demeyer, T. Mens, M. Wermelinger. *Towards a Software Evolution Benchmark*, International workshop on principles of software evolution, 2001

[French 99] V.A. French. *Establishing Software Metric Thresholds*. International Workshop on Software Measurement (IWSM’99), 1999

[Ducasse 99] S. Ducasse, M. Rieger, S. Demeyer. *A Language Independent Approach for Detecting Duplicated Code*, icsm, p. 109, 15th IEEE International Conference on Software Maintenance (ICSM'99), 1999.

[Gray 96] A.R. Gray, S.G. MacDonell. *A comparison of techniques for developing predictive models of software metrics*, Elsevier, 1996

[Schneidewind 92] N.F. Scneidewind. *Methodology For Validating Software Metrics*, IEEE Transactions on Software Engineering, 1992

[Samoladas 04] Samoladas, I., Stamelos, I., Angelis, L., and Oikonomou, *Open source software development should strive for even greater code maintainability*, Communications of the ACM 47, 10 (Oct. 2004), 83-87.

[Kettelerij 05] R. Kettelerij, B.G. Prijn. *Detection Of Duplicated Code In Large Software Systems*, Graduation report University of Arnhem and Nijmegen, 2005

> *This work describes the development of a language independent duplicated code detection tool. A major design goal of this tool is performance. This tool is used in this study to calculate code duplication.*

[Meijles 05] J. Meijles. *Analysis of designers’ work*, Master thesis University of Amsterdam, 2005

[Fenton 99] N.E. Fenton, M. Neil. *Software Metrics: successes, failures and new directions*, Elsevier, 1999

[Fenton 97] N.E. Fenton, S.L. Pfleeger. *Software Metrics Second Edition*, PWS Publishing Company, 1997

[Menzies 02] T. Menzies, J.S. Fenton, Justin S. Di Stefano, M. Chapman, K. McGill. *Metrics That Matter*, Proceedings of the 27th Annual NASA Goddard/IEEE Software Engineering Workshop, 2002

[Sim 03] Susan Elliott Sim, Steve Easterbrook, Richard C. Holt, "Using Benchmarking to Advance Research: A Challenge to Software Engineering," icse, p. 74, 25th International Conference on Software Engineering (ICSE'03), 2003.

[Cui 06] Qingguang Cui, Matthew O. Ward, Elke A. Rundensteiner, Jing Yang, "Measuring Data Abstraction Quality in Multiresolution Visualization", To appear in IEEE Symposium on Information Visualization 2006 (InfoVis'06), 2006 http://davis.wpi.edu/~xmdv/docs/infovis06_measure.pdf

[SIG] Software Improvement Group, http://www.sig.nl

[Checkstyle] Checkstyle, http://checkstyle.sourceforge.net

[SWEBOK 04] Guide to the Software Engineering Body of Knowledge, http://www.swebok.org

[Spring] Spring Framework, http://www.springframework.org

[Hibernate] Hibernate, http://www.hibernate.org

[Jakarta] The Apache Software Foundation, Jakarta Commons, http://jakarta.apache.org/commons