Investigating the relation between static and dynamic coupling metrics and the fault-proneness of object-oriented software systems

Version of July 25, 2006

source: http://www.opte.org

Atam Gangaram Panday
Investigating the relation between static and dynamic coupling metrics and the fault-proneness of object-oriented software systems

THESIS

submitted in fulfillment of the requirements for the degree of

MASTER OF SCIENCE

In

SOFTWARE ENGINEERING

by

Atam Gangaram Panday
Investigating the relation between static and dynamic coupling metrics and the fault-proneness of object-oriented software systems

Author: Atam Gangaram Panday
Student id: 1099809
Email: Atamgp@gmail.com

Thesis Committee:

- Prof. Dr. A. van Deursen, Faculty EEMCS, TU Delft
- Ir. F. Ververs, Faculty EEMCS, TU Delft
- Drs. J.W.J. Heijnsdijk, Faculty EEMCS, TU Delft
ACKNOWLEDGEMENTS

Most of the work regarding this thesis work was carried out at The CUSAT University at Cochin, India. I like to thank Mr. David Peter for arranging every thing for me in Cochin and Dr. Sumam Mary Indicula who was supervising me there. My thanks also go to Dr. Asha Gopalakrishan from the Statistics Department of the CUSAT University who arranged access to statistic related books and information. I am grateful to Mr. Jan de Vries of the Computer Science department at the TUDelft University for making this foreign project possible from the TUDelft side. My final thanks go to Ir. Frans Ververs who was my supervisor for this project and advisor at the TUDelft University.

I like to thank my parents for supporting me and giving me the opportunity to study Computer Science in the Netherlands. This is most likely one of the biggest and best decisions taken in my life.

Atam Gangaram Panday
Delft, the Netherlands
July 25, 2006
CONTENTS

ACKNOWLEDGEMENTS .................................................................................................................. 7

CONTENTS .......................................................................................................................................... 9

ABBREVIATIONS ............................................................................................................................... 11

1 INTRODUCTION .............................................................................................................................. 13

1.1 Thesis Overview ............................................................................................................................ 13
1.2 Problem Definition ...................................................................................................................... 14
1.3 Contribution .................................................................................................................................. 14
1.4 Thesis Organization ...................................................................................................................... 15

2 SOFTWARE METRICS AND DESIGN QUALITY ........................................................................... 17

2.1 Software Metrics as a Part of Software Engineering ................................................................... 17
2.2 Types of Software Metrics .......................................................................................................... 19
2.3 Software Metrics and Object-Oriented Design ......................................................................... 22

2.3.1 Concepts in Object-Oriented Design ...................................................................................... 22
2.3.2 Object-Oriented Metrics History ............................................................................................ 24
2.3.3 The need for dynamic metrics ................................................................................................ 26

2.4 Dynamic Coupling ...................................................................................................................... 27

2.4.1 A Dynamic Coupling Metrics Suite ....................................................................................... 27
2.4.2 Dynamic Coupling Measures from early stage of lifecycle .................................................... 37

3 DEFINITION OF SOME NEW OBJECT-ORIENTED METRICS ..................................................... 39

3.1 Some Additional Dynamic Coupling Measures ......................................................................... 39
3.2 Introducing Neighbor-Coupling ................................................................................................ 40
3.3 Defining and Validating Some Neighbor-Coupling Measures ................................................ 43
3.4 The Yet Missing Indicator of Usage of Object-Oriented Features: DII ..................................... 49

4 MEASURING DYNAMIC COUPLING IN JAVA .......................................................................... 51

4.1 Java and the Java Virtual Machine ............................................................................................. 51

4.1.1 The java programming Language ....................................................................................... 51
4.1.2 The Java Virtual machine & its components ....................................................................... 52
4.1.3 Execution of a java program ............................................................................................... 52

4.2 Java Debugger Interface and Bloat ............................................................................................ 53
4.3 JREV: Tracing Java Program Execution ..................................................................................... 54
4.4 Velocity ....................................................................................................................................... 55
4.5 Tomcat ........................................................................................................................................ 56

5 ANALYSIS BASE AND HYPOTHESIS ...................................................................................... 57

5.1 Dependent Variable ..................................................................................................................... 58
5.2 Independent Variables ............................................................................................................... 58
5.3 Hypotheses ................................................................................................................................... 60

6 DATA ANALYSIS APPROACH .................................................................................................. 63

6.1 Descriptive Statistics .................................................................................................................. 63
6.2 Outlier Analysis ........................................................................................................................ 65

6.2.1 Univariate outlier detection ................................................................................................... 65
6.2.2 Multivariate outlier detection ............................................................................................... 67

6.3 Principal Components Analysis and Varimax Rotation ........................................................... 69
## Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CBO</td>
<td>Coupling Between Objects</td>
</tr>
<tr>
<td>CBOExp</td>
<td>CBO, but only accounting for export coupling</td>
</tr>
<tr>
<td>CBOImp</td>
<td>CBO, but only accounting for import coupling</td>
</tr>
<tr>
<td>DII</td>
<td>Dynamic Invocation Indicator</td>
</tr>
<tr>
<td>EC_CC</td>
<td>Dynamic Export Coupling (entity of measurement = class, strength = distinct classes)</td>
</tr>
<tr>
<td>EC_CCC</td>
<td>Dynamic Export Coupling (entity of measurement = class, strength = distinct Class-to-Class)</td>
</tr>
<tr>
<td>EC_CM</td>
<td>Dynamic Export Coupling (entity of measurement = class, strength = distinct methods)</td>
</tr>
<tr>
<td>EC_OC</td>
<td>Dynamic Export Coupling (entity of measurement = object, strength = distinct classes)</td>
</tr>
<tr>
<td>EC_OCC</td>
<td>Dynamic Export Coupling (entity of measurement = object, strength = distinct Class-to-Class)</td>
</tr>
<tr>
<td>EC_OM</td>
<td>Dynamic Export Coupling (entity of measurement = object, strength = distinct methods)</td>
</tr>
<tr>
<td>IC_OC</td>
<td>Dynamic Import Coupling (entity of measurement = object, strength = distinct classes)</td>
</tr>
<tr>
<td>IC_OCC</td>
<td>Dynamic Import Coupling (entity of measurement = object, strength = distinct Class-to-Class)</td>
</tr>
<tr>
<td>IC_OM</td>
<td>Dynamic Import Coupling (entity of measurement = object, strength = distinct methods)</td>
</tr>
<tr>
<td>IC_CC</td>
<td>Dynamic Import Coupling (entity of measurement = class, strength = distinct classes)</td>
</tr>
<tr>
<td>IC_CCC</td>
<td>Dynamic Import Coupling (entity of measurement = class, strength = distinct Class-to-Class)</td>
</tr>
<tr>
<td>IC_CM</td>
<td>Dynamic Import Coupling (entity of measurement = class, strength = distinct methods)</td>
</tr>
</tbody>
</table>
NAImp: Number of non-inherited Attributes Implemented
NG_CBO: Neighbor Coupling thru CBO
NG_CBOExp: Neighbor Coupling thru CBOExp
NG_CBOImp: Neighbor Coupling thru CBOImp
NG_EC_CCC: Neighbor Coupling thru EC_CCC
NG_EC_OCC: Neighbor Coupling thru EC_OCC
NG_IC_CCC: Neighbor Coupling thru IC_CCC
NG_IC_OCC: Neighbor Coupling thru IC_OCC
NMImp: Number of non-inherited Methods Implemented
PCA: Principal Component Analysis
PC: Principal Component (PCA)
Chapter 1

Introduction

1.1 Thesis Overview

Fault-proneness prediction-model construction is not new. There has been done much research within this area. These models take as input some software metrics values and produce their outcome, which is a prediction regarding the faultiness (or some other external software attribute) of the regarded software class, assuming object-oriented technology is used as the base of the research.

The last years, dynamic metrics have been a huge subject to investigation whether they are more precise in prediction models (e.g. fault-proneness) for object-oriented software. Many different dynamic metrics were defined of which not all were validated. In this thesis the dynamic coupling suite of Arisholm [3, 4 and 5] will be used in addition to some new defined dynamic coupling metrics and (static and dynamic) neighbor coupling metrics.

In much of the research done within this area of computer science, the way the prediction-model is constructed, fault/metric data is selected and the results are interpreted differs. This makes it difficult to interpret and compare results among different projects. Therefore the steps from data definition to data analysis are done regarding a framework mostly part of the one used by L.C. Briand [14 and 15]. By using such a framework, the results of this thesis can be interpreted more consistently and therefore be of a bigger contribution.

A yet missing part in object-oriented software metrics research is the differences between two or more software systems regarding their usage of object-oriented concepts/features. It are these aspects which would make dynamic metrics different from their static equals. In order to fill this gap the DII (dynamic invocation indicator) metric is defined and used to explain the differences in analysis outcome regarding the two considered software systems, Velocity and Tomcat, who’s DII will be used in explaining the differences noticed regarding their individual relation between their coupling metrics and their fault-proneness.
1.2 Problem Definition

The problem definition of this Thesis is to examine the predictability of the fault-proneness of object-oriented software of dynamic coupling metrics versus static coupling metrics, not disregarding the fact that object-oriented software systems may differ in their level of usage of object-oriented features, which may have an impact on the quality of dynamic measures as an indicator. For dynamic coupling metrics, those of Arisholm [3, 4 and 5] will be used, together with some newly defined metrics (chapter 3).

1.3 Contribution

The main contribution in this thesis is stating that object-oriented software differs in their usage of object-oriented features which is the basis for object-orientated metrics, especially dynamic ones. This causes differences when analyzing dynamic metrics of different object-oriented software systems. To help dealing with this phenomenon a new metric DII, dynamic invocation indicator, is defined. Besides DII, a few dynamic coupling metrics are defined along with a new coupling type: Neighbor Coupling, which is also considered in the analysis carried out. A tool “Jrev” was developed for extracting all these metrics along with some static coupling metrics and some size metrics from an input Java software system. This tool will be made available for others. The final contribution of this thesis project is supporting the usage of a framework in analyzing metrics data, which will eventually (hopefully) lead to some standard in software metrics analysis in order for research in this area being more consistent and easier for interpretation.
1.4 Thesis Organization

This thesis is organized in 8 chapters. The Table below gives an overview of what these chapters contain.

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><strong>Introduction:</strong> provides an overview of the work done in this thesis project together with a problem definition and an overview of the thesis.</td>
</tr>
<tr>
<td>2</td>
<td><strong>Software Metrics and Design Quality</strong> presents an introduction to software metrics as a part of Computer Science and gives an overview of the dynamic coupling metrics suite of Arisholm, which will be used throughout this thesis.</td>
</tr>
<tr>
<td>3</td>
<td><strong>Definition of some new Object-Oriented Metrics</strong> gives the definition of some additional dynamic coupling metrics and introduces Neighbor Coupling and the DII (Dynamic Invocation Indicator) metric.</td>
</tr>
<tr>
<td>4</td>
<td><strong>Measuring Dynamic Coupling in Java</strong> gives an overview of the Java programming language, JDI (Java Debugging Interface) and the Bloat library which are used by the developed tool “Jrev” in order to trace execution of Java programs and compute out of this collected data the metrics mentioned in this thesis. Two Java software systems, Velocity and Tomcat, and investigated in this thesis.</td>
</tr>
<tr>
<td>5</td>
<td><strong>Analysis base and Hypotheses</strong> defines the base for this project, the dependent and independent variables and the hypotheses.</td>
</tr>
<tr>
<td>6</td>
<td><strong>Data Analysis Approach</strong> explains the used framework for data analysis done in his project.</td>
</tr>
<tr>
<td>7</td>
<td><strong>Data collection and analysis</strong> shows the results of the steps of the previously explained framework for data analysis regarding Velocity and Tomcat together with an evaluation of the previously stated hypotheses (chapter 5).</td>
</tr>
<tr>
<td>8</td>
<td><strong>Conclusions and Future work</strong> lists the derived conclusions of this project and the suggested future work regarding this area of research.</td>
</tr>
</tbody>
</table>

Table 1.1: Thesis Organization
Furthermore a reference list is given followed by Appendix A which contains the design of the Jrev tool, and Appendix B which contains the public API of Jrev.
Chapter 2

Software Metrics and Design Quality

In this chapter Software Metrics will be introduced. The importance of this part of Science will be explained, and a brief history will be given. The distinction between different kinds of metrics will be also explained as is the concept of a model. Further, a brief overview of object-oriented technology and its relation with software metrics will be given, together with concepts like static and dynamic metrics. Because this thesis focuses much on dynamic coupling metrics, these will be introduced in the last section.

2.1 Software metrics as a part of Software Engineering

The term software engineering was coined in 1968 as a response to the desolate state of the art of developing quality software on time and within the budget. Software developers were not able to set concrete objectives, predict the resources necessary to attain those objectives, and manage the customers’ expectations [25]. When developing software, one is often faced with ill-defined problems, partial solutions, and has to rely on empirical methods to evaluate solutions. To cope with this problem of building and delivering complex software systems on time, the software engineering discipline was brought into live, to make software engineers able to build high quality products, and integrating them under time and budget constrains. Analyzing this discipline, software engineering can be defined as a modeling activity, thru which engineers deal with complexity.

In [25] software engineering is defined as the term used to describe the collection of techniques concerned with applying an engineering approach to the construction of software products. By engineering approach is meant managing, costing, planning, modeling, analyzing, designing, implementing, testing and maintaining. These activities together with tools and techniques to support and integrate them have long been seen as the solution to poor quality software systems delivered late and over-budget, which are becoming common these days. In summary, software production is often out of control. It has been suggested very long, that this is the case because we do not measure software. According to Tom DeMarco:
“You cannot control what you cannot measure.”

Measurement outside software engineering has been in use long before software measures were introduced. Measurements in economic systems determine price and pay increases. Measurements in medical systems enable the diagnosis of specific illnesses. A more general definition of measurement is:

*Measurement is the process by which numbers or symbols are assigned to attributes of entities in the real world in such a way as to describe them according to clearly defined rules.*

The attribute being measured is the feature or property of the entity we are interested in. The assignment of numbers or symbols to these attributes creates a way to interpret, describe and compare measurements according to their defined rules.

As today’s software applications become more complex and software failure is more critical, potentially resulting in economic damage or even threatening the health or lives of human beings, a means of effectively measuring the quality of software products is needed. Where software engineering was initially brought into live to deal with problems concerning software development, it nowadays also has to incorporate a way to analyze the resulting product in terms of quality, complexity, reliability and other useful aspects, which are called software metrics. Software metrics can be loosely defined as activities concerned with measurement in software engineering. Software metrics can provide useful information to project managers and software developers by providing a means of measuring the complexity of a software product.

Although using software metrics will give one more control on the development of software, it is not enough. For measures taking place, specific motivations are needed. There have to be clear objectives or goals, and it is these which will determine the kinds of entities and attributes which must be measured. To support this, each defined software measure has to be clearly assigned its type of measure, as explained in the next section, and has to be validated, which will be done in the next chapter for some newly defined measures. Being aware of this information regarding a certain measure, one should be able to know its usage, in what circumstances that measure is valid, and how to interpret the results.
2.2 Types of Software metrics

In software there are three classes of entities whose attribute we may wish to measure [25]. These are:

- **Processes** which are any software related activities; these normally have a time factor.
- **Products** which are any artifacts, deliverables or documents which arise out of the process.
- **Resources** which are the items which are inputs to processes.

Anything that we are ever likely to want to measure or predict in software is an attribute of some of the above three classes.

A distinction can be made between attributes of these entities, which are internal or external:

- **Internal attributes** of a product, process or a resource are those which can be measured purely in terms of the product, process or resource itself.
- **External attributes** of a product, process or resource are those which can be only measured with respect to how the product, process and or resource relates to its environment.

External attributes cannot be measured directly, this means that there measurement involves the measurement of one or more other attributes. It are these attributes that software managers would like to measure and predict. For example, software managers would like to know the cost-effectiveness of some processor or the productivity of their personnel. Other examples of external attributes include usability, reliability, fault-proneness, maintainability and testability. Unfortunately, external attributes are by their very nature the most difficult to measure. Moreover, there are usually no agreed definitions; attributes like quality are so general that they are almost meaningless.

Internal attributes, on the other hand, can be measured directly. That means that measuring these kinds of attributes does not require measuring another attribute or does not depend on another attribute. A simple example is the number of non-inherited attributes declared in a software class,
which is named NAImp. This is an object-oriented measure, which will be used later on, and the entity of measurement is a software class, which is a product entity.

**Processes** are software related activities which are usually related to time. Example are *constructing a specification document* or *developing a software system from requirements capture through to the release to the customer*. There are only a limited number of internal attributes of interest which are directly measurable for processes:

- *Time*, i.e. duration of the process
- *Effort* associated with the process
- *Number of incidents of a specified type* arising during the process

**Products** are taken to mean deliverables, artifacts or generally documents which arise from the software lifecycle. Examples include specification and design documents at various levels of detail, representations of the source or object code (e.g. a software class), and test strategy documents.

Examples of internal attributes are length, functionality, modularity, reuse, redundancy and syntactic correctness for specification documents, formal designs and code. For formal designs and code additionally we might be interested in various types of structuredness, coupling and cohesiveness.

Examples of external attributes are:

**Reliability** of program code. This is dependent on both the machine on which the program is running and on its mode of operation usage (which also includes as implicitly understood specification).

**Understandability** of a specification document. This is dependent on the person who is trying to understand it.

**Maintainability** of source code. This is dependent on both the person performing the maintenance and any tools available to support this.
Other well-known examples include usability, integrity, efficiency, testability, reusability, portability, interoperability and fault-proneness, of which the last one will be investigated further in this thesis.

**Resources** are the miscellany of items considered as inputs for software production. Examples include personnel (individual or teams), materials, tools (software, hardware) and methods. An attribute of great interest which is relevant to all of these types of resources is cost, which is considered in many situations to be dependent on a number of attributes in addition to the one which is most easily measurable, namely (market) price.

When we talk about measuring something we normally mean that we wish to assess some entity which already exists. However, in many circumstances we might also wish to predict an attribute of some entity which does not yet exist. For example although we can only truly assess the reliability of a software system once it is operational, we also try to **predict** its likely reliability on the basis of our knowledge of the system when under development.

To better understand prediction, and the goal of this thesis which I will explain in later chapters, the definition of a **model** is needed:

> A model is an abstract representation of an object.

A model, with respect to software metrics, can be an abstract representation of the various products, processes and resources. These are needed to define measures unambiguously. Such models must capture the attributes being measured. Another kind of model, with respect to software metrics, is an abstract representation of relationships between attributes and entities. These models usually relate two or more measures in a mathematical formula.

A simple example of the latter one is:

\[ m = \frac{x}{a} \]

Where \( x \) is a variable representing a measure if source-code program length, \( m \) is a measure of number of hard copy pages for source-code programs, and \( a \) is a constant.
The extend to which such a model is being used for assessment, as opposed to solving a prediction problem depends on how much is known about the parameters of the model.

2.3 Software Metrics and Object-Oriented Design

As object oriented (OO) analysis and design techniques became more widely used in industry, the demand on correctly assessing the quality of object-oriented designs substantially increases. In the beginning existing metrics were used for object-oriented software. Looking at the concepts behind the object-oriented paradigm may provide an indication as to why these metrics were inappropriate. A brief overview of some proposed object-oriented metrics will follow there after, finishing this section with the distinction between static and dynamic metrics for object-oriented software.

2.3.1 Concepts in Object-Oriented Design

In one way, object orientation can be seen as a development of the practice in structured system analysis and design methods of structuring the data that a computer application needs to access and manipulate by organizing it so that it models the corresponding objects in the ‘real world’ [33]. These entities may be physical entities, such as a person or a car, or more conceptual objects, such as a bank account.

The fundamental concept in object-orientation is that of an object. An object encapsulates a part of its state in data members, descriptions of manipulations in member functions and exhibits a well-defined behavior. A class is a blueprint for an object; it provides a specification and implementation of the properties of its object instances. There exists a distinction between the specification of a class and its implementation. The implementation comprises the internal view and is hidden from other classes. This is to enforce information hiding. However this may be violated through class inheritance.

The major concepts associated with the OO paradigm include the following:

- **Abstraction** - the identification of the common characteristics of a group of objects in order to form a class with the shared characteristics. Another word for abstraction often
used is granularity. Using abstraction permits selective information hiding based on scale issues. Classification is one particular form of abstraction by which individuals (objects) of similar/identical characteristics are grouped together in a common class [30]. Another level of abstraction would be software components.

- **Encapsulation** – a development of the idea of information hiding. An object is only accessible through a predetermined interface. Thus the precise way in which variables are structured and the detail of the mechanisms by which they are actually manipulated can be concealed from the user of that object. As long as the behavior of the interface is unchanged, the underlying mechanisms can be changed at will.

- **Inheritance** – inheritance allows programmers to define classes incrementally by reusing previously defined classes as the basis for the new ones. If a class A inherits from a class B, then A includes all the attributes and methods declared in class B, including those which B may inherit from another class C. Here class B is the super class of class A, and class A is a subclass of class B. Also, by using this technique, a class can be divided into subclasses. Where the base class from which the subclasses inherit includes the functionality needed by all these subclasses, and each subclass is extended with its specific features. One motivation for the use of inheritance is the hope that it might lead to economic savings through the increased code reuse that the ability to inherit methods might bring. If a class can inherit from more than one class directly, multiple-inheritance is supported. For simplicity sake, and because multiple-inheritance is not that much related to the main goal of this thesis, multiple inheritance will not be considered in the rest of the thesis.

- **Polymorphism** – a method belonging to a class would be executed when a request to carry out that particular service is made to an object belonging to that class. Polymorphism allows the same type of request to be made to different classes, if these classes inherit from the class the method is declared in, and a possibility for each class to deal with that request in its own way, by overwriting the definition of that method in its specification/implementation.

Overall, these concepts should result in support to deliver products to market more quickly and to provide higher-quality products with lower maintenance costs. But without the right
disciplines/techniques (software engineering) and metrics development and management, such software products may get out of hand.

2.3.2 Object-Oriented Metrics History

Compared to structural software development, object-orientation is based on completely different concepts, as described in the previous sub-section. Intuitively, it would seem implausible for traditional metrics to be applied to object-oriented software. This is true for most metrics [17, 25, 39 and 43], however, in the end, all software, regardless the way it has been constructed, ends up as machine instructions (with the java language as an exception whose executable is interpreted). At this level, traditional measures should not be a problem. On the other hand, the main entity of interest now is the class instead of a module. Also, a big difference with traditional software development is that object-oriented design puts greater emphasis on the design phase of the software development lifecycle, whereas structured development methods focus on the implementation phase, which is also why object-oriented metrics aimed at the design of an object-oriented system are more suited to the object-oriented paradigm [34].

As the existing metrics/non-OO metrics for structured development were subject to general criticism and easily seen as not supporting object-oriented concepts, a sheer volume of measurements have been suggested. However, a suite of metrics proposed by Chidamber and Kemerer (1991) [7 and 34] to measure the unique aspects of the object-oriented approach has gained much attention. Their metrics are aimed at the design of an object-oriented system and one of them, Coupling Between Objects, will be used in this thesis:

The **Coupling Between Objects** measure was originally defined as “a count of the number of non-inherited related couples with other classes”. Two objects are deemed to be coupled if they ‘act upon’ one another, in other words, if an object of one class uses the methods or instance variables of the other. If a method declared in one class uses a method or instance variable in another class, this pair of classes are said to be coupled since all objects instantiated from the same class are deemed to have the same properties.

However, Chidamber and Kemerer later revised their definition of CBO. For a class C, CBO is a measure of the number of other classes to which it is coupled. They amended their previous definition to include coupling due to inheritance, but they provided no explanation for this.
A point to note is that this metric is not transitive. Suppose a class A uses methods or instance variable from a class B and class B uses methods or instance variables from a class C. A is said to be coupled to B and B is said to be coupled to C, but this does not imply that A is coupled to C.

Chidamber and Kemerer maintained that in order to promote encapsulation of classes, that is the inclusion within a program of all the resources the class needs to function, it is desirable to reduce coupling between structures. The excessive coupling between structures outside of the inheritance hierarchy is detrimental to modular design and prevents reuse. This metric was considered to be useful in determining the effort involved in testing the various parts of a software system [12]. The greater number of couples a class has with other classes, the more rigorous the testing process would need to be.

For comparison with dynamic coupling metrics, CBO will be included in this thesis for further analysis. Along CBO, its two variants, CBOImp and CBOExp will also be considered. Where CBO accounts for both, import and export coupling between classes, CBOImp only accounts for import coupling and CBOExp only for export coupling.

Other metrics part of the suite of Chidamber and Kemerer are The Depth of Inheritance Tree (DIT), the Number of Children (NOC), the Response for a Class (RFC), Weighted Methods per Class (WMC) and the Lack of Cohesion in Methods (LCOM).

Where earlier the LOC (Lines of Code) was a very famous size metric, the two size metrics used in this thesis are NAImp and NMImp. NAImp counts per class the number of non-inherited attributes of that class. NMImp counts the number of non-inherited methods of that class.

For an overview of much more suggested object-oriented metrics, see Hughes[33] and Henderson [30], [39, 43 and 47].
2.3.3 The need for dynamic metrics

Regardless of the attribute considered, most metrics so far have been defined and collected based on a static analysis of the design or code [3, 4, and 5]. They have, on a number of occasions, proven to be accurate predictors of external quality attributes, such as fault-proneness, ripple effects after changes, and changeability. However, many of the systems that have been studied showed little inheritance and, as a result, limited use of polymorphism and dynamic binding. As the use of object-oriented design and programming matures in industry, we observe that inheritance and polymorphism are used more frequently to improve internal reuse in a system and facilitate maintenance. Though no formal survey exists on this matter, this is visible when analyzing the increasing number of open source projects, application frameworks, and libraries.

The problem is that static measures may lose precision as more intensive use of inheritance and dynamic binding occurs. This is expected to result in poorer predictive accuracy of the quality models that utilize static measures. Measures which are mostly influenced by inheritance, polymorphism and dynamic binding are coupling and cohesion measures.

Let us take an example, as illustrated in Figure 2.1 which I took from Arisholm [3, 4 and 5], to clarify the issue at hand. Due to inheritance, the class of the object sending or receiving a message may be different from the class implementing the corresponding method. For example, let object a be an instance of class A, which is inherited from ancestor A'. Let A' implement the method mA'. Let object b be an instance of class B, which is inherited from ancestor B'. Let B' implement the method mB'. If object a sends the message mB' to object b, the message may have been sent from the method source mA' implemented in class A' and processed by a method target mB' implemented in class B'. Thus, in this example, message passing caused two types of coupling: (1) object-level coupling between class A and class B (i.e., coupling between instances of A and B), and (2) class-level coupling between class A' and B'. The code may very well show statements where an object of type A invokes from mA' method mB' on an object of type B. However, to assume, through static code analysis, that there is class-level coupling between A and B as a result, is simply inaccurate. Both types of coupling, at the class and object levels, need to be captured accurately to address certain applications and must be investigated.
2.4 Dynamic Coupling

In this sub-section the proposed dynamic coupling measures by Arisholm [3, 4 and 5] will be explained. As dynamic coupling measures become more famous, more of these type of measures have been defined [47]. There after a sub selection of these metrics will be taken for usage in the rest of this thesis, based on the need to measure from the design entity.

2.4.1 A Dynamic Coupling Metrics Suite

A set of coupling measures was proposed by Arisholm, referred to as dynamic coupling measures, which are defined on the analysis of run-time object interaction. They can be collected through a dynamic analysis of the code, that is, by executing the code and saving information regarding the messages that are being sent among objects at run-time. In chapter 4 a tool, Jrev, will be introduced which makes this possible.

I first distinguish different types of dynamic coupling measures. Then, based on this classification, we provide both informal and formal definitions, using a working example to illustrate the fundamental principles. The following is mostly taken from [3, 4 and 5], which therefore provide a better overview of these metrics.
Classifying Coupling Measures

Three decision criteria were used to define and classify dynamic coupling measures.

1. *Entity of measurement* - Since dynamic coupling is based on dynamic code analysis, coupling may be measured for a class or one of its instances. The *entity of measurement* may therefore be a class or an object.

2. *Granularity* - Orthogonal to the entity of measurement, dynamic coupling measurement can be aggregated at different levels of *granularity*. With respect to dynamic *object* coupling, measurement can be performed at the object level, but can also be aggregated at the class level, i.e., the dynamic coupling of all instances of a class is aggregated. In practice, even when measuring object coupling, the lowest level of granularity is likely to be the class, as it is difficult to imagine how the coupling measurement of objects could be used. Alternatively, all the dynamic coupling of objects involved in an execution scenario can be aggregated. We can also measure the dynamic object coupling in entire use cases (i.e., sets of scenarios), sets of use cases, or even an entire system (all objects of all use cases). In the case where the entity of measurement is a class, the aggregation scale is different as we can aggregate dynamic *class* coupling across an inheritance hierarchy, a subsystem, a set of subsystems, or an entire system.

3. *Scope* - Another important source of variation in the way one can measure dynamic coupling is the *scope* of measurement. This determines which objects or classes, depending on the entity of measurement, are to be accounted for when measuring dynamic coupling. For example, we may want, depending on the application context, to exclude library and framework classes, which will be the case in this project.
At the object level, we may want to exclude certain use cases modeling exceptional situations (e.g., error conditions) or objects that are instances of library or framework classes. At the very least, one may want to distinguish the different types of coupling taking place in these different categories. The choices made regarding the entity, granularity, and scope of measurement depend on how one intends to apply dynamic coupling. Such choices form a classification of dynamic coupling measures that is summarized in Table 2.1.

**Definitions**

Before listing the dynamic coupling measures, below the formal framework used by Arisholm is given which provides precise and unambiguous definitions. Not only do such definitions ensure that the reader understands the measures precisely, but they are also easily amenable to the analysis of their properties and facilitate the development of a dynamic analyzer by providing precise specifications. A set of generic definitions is given that are based on the data model in Figure 2.2, which models the type of information to be collected. Each class and association in the class diagram corresponds to a set and a mathematical relation, respectively. The inheritance relationship corresponds to a set partition. Based on this, the measures using set theory and first order logic were defined.
Figure 2.2: Class Diagram Capturing a Data Model of the Dynamic Analysis

A few details of the class diagram in Figure 2.2 need to be discussed. For example, methods are defined in a class, method invocations consist of a caller method in a source class and a callee method in a target class. Some of the key attributes are shown. One notable detail is that the line number where the target method is invoked is an attribute of a message that serves to uniquely identify it. This is necessary, because the same target method may be invoked in different statements and control flow paths in the same source method. Messages bearing those different invocations are considered distinct, because they are considered to provide different contexts of invocation for the method. Furthermore, associations with role names caller, source and sender should show an \{exclusive or\} constraint dependency to associations with role names callee, target, and receiver, respectively. These constraints are not shown to avoid cluttering the diagram but are important as in our context; distinct methods, classes and objects must be involved in the links corresponding to those associations. In other words, in the context of our coupling measurement, method invocations are linked to two distinct class instances and two distinct method instances and messages involve two distinct objects. As expected, method invocations between classes are differentiated from messages between objects. A method name and signature
uniquely identifies a method in the context of a specific class and a method invocation must be clearly linked to a method. This is why MethInvocation has associations with both Class and Method.

**Sets**

Next, the basic sets are given upon which some definitions were built. These sets were derived from the data model in Figure 2.2:

- **C**: Set of classes in the system. C can be partitioned into the subsets of application classes (AC), library classes (LC), and framework classes (FC). Some of these subsets may be empty, \( C = AC \cup LC \cup FC \) and \( AC \cap LC \cap FC = \emptyset \). Distinguishing such subsets may be important for defining the scope of measurement, as discussed above.

- **O**: Set of objects instantiated by the system while executing all scenarios of all use cases (including exceptional use cases, e.g., treating error conditions, which are usually modeled as use cases extending base use cases).

- **M**: Set of methods in the system (as identified by their signature).

- Lines of code are defined on the set of natural numbers \( \mathbb{N} \)

**Relations**

The mathematical relations on the sets which are fundamental to the definitions of the dynamic measures are listed below.

- D and A are relations onto \( C (\subseteq C \times C) \). D is the set of descendent classes of a class and A is the set of ancestors of a class.
• ME is the set of possible messages in the system: $ME \subseteq O \times M \times N \times O \times M$. Indicated by the domain of ME, a message is described by a source object and method sending the message, a line of code ($N$), and a target object and method.

• IV is the set of possible method invocations in the system: $IV \subseteq M \times C \times M \times C$. An invocation is characterized by the invoking class and method and the class and method being invoked.

• Other binary relations will be used in the text and their semantics can be easily derived from their domain and are denoted $R_{\text{Domain}}$. For example, $R_{MC} \subseteq M \times C$ refers to methods being defined in classes, a binary relation from the set of methods to the set of classes.

*Consistency Rule*

The relations IV and ME play a fundamental role in all these measures. In practice, an analysis of sequence diagrams or a dynamic analysis of the code allows us to construct ME. From that information, IV must be derived, but this is not trivial as polymorphism and dynamic binding tend to complicate the mapping. The consistency rule below [3, 4 and 5] specifies the dependencies between the two relations and can be used to develop algorithms that derive IV from ME:

$$
(\exists (o_1, c_1), (o_2, c_2) \in R_{OC}) (\exists l \in N) (o_1, m_1, l, o_2, m_2) \in ME \Rightarrow \\
(\exists c_3 \in A(c_1) \cup \{c_1\}, c_4 \in A(c_2) \cup \{c_2\}) \\
((m_1, c_3) \in R_{MC} \land ((\forall c_5 \in A(c_1) - \{c_3\}) (m_1, c_5) \in R_{MC} \Rightarrow c_5 \in A(c_3))) \land \\
((m_2, c_4) \in R_{MC} \land ((\forall c_6 \in A(c_2) - \{c_4\}) (m_2, c_6) \in R_{MC} \Rightarrow c_6 \in A(c_4))) \land \\
(m_1, c_3, m_2, c_4) \in IV
$$

*Working Example*
I now show a working example from [3, 4 and 5], as shown in Figure 2.3, which illustrates the definitions above. Though it is assumed that these measures are collected through static and dynamic analysis of code, UML is used to describe a fictitious example, because it is more legible than pseudo code. This example was designed to illustrate the subtleties arising from polymorphism and dynamic binding. Other aspects, such as method signatures, have been intentionally kept simple to focus on polymorphism and dynamic binding.

![Figure 2.3 Working Class Diagram Example (UML notation)](image)

The following sets can be derived from Figure 2.3:

- \( C = \{c_1, c_2, c_3, c_4, c_5\} \)
- \( M = \{m_1, m_2, m_3\} \)
- \( RMC = \{(m_1, c_1), (m_2, c_2), (m_3, c_3)\} \)

In order to derive other relevant sets and relations, the sequence diagrams is given in Figure 2.4, where each message is numbered. As this fictitious example is represented with UML diagrams, objects are referred to by using the sequence diagram number where they appear and their own identification number (i.e., \( SDi:object\ id \)). Similarly, the line of code of the method invocation is denoted in message tuples as \( l(SDi:message\ id) \). In the example, it is assumed that the line of code of the method invocations \( m_3() \) in messages \( SD1:1.1, SD1:1.2 \) and \( SD1:1.3 \) are different. Furthermore, since the sequence diagrams do not specify the sender object, source class and source method of the method invocations \( m_1() \) in messages \( SD1:1 \) and \( SD2:1 \), the
example sets derived below account for only the four (completely specified) messages SD1:1.1, SD1:1.2, SD1:1.3 and SD2:1.1:

\[ O = \{SD1:1, SD1:2, SD1:3, SD2:1, SD2:2\} \]

\[ ROC = \{(SD1:1, c1), (SD1:2, c4), (SD1:3, c5), (SD2:1, c1), (SD2:2, c2)\} \]

\[ ME = \{(SD1:1, m1, l(SD1:1.1), SD1:2, m3), (SD1:1, m1, l(SD1:1.2), SD1:3, m3), (SD1:1, m1, l(SD1:1.3), SD1:3, m3), (SD2:1, m1, l(SD2:1.1), SD2:2, m2)\} \]

\[ IV = \{(m1, c1, m3, c3), (m1, c1, m2, c2)\} \]

*Definitions of Measures*

The measures are all defined as cardinalities of specific sets. Those sets are defined below and are given self-explanatory names, following the notation summarized in Table 2.2. First, as mentioned above, a differentiation is made concerning the cases where the entity of measurement is the object or the class. Second, as in previous *static* coupling frameworks, a differentiation of *import* from *export* coupling is made, that is the *direction* of coupling for a class or object. For example, there is a made a difference whether a method executed on an object calls (imports) or is called by (exports) another object’s method. Furthermore, orthogonal to the entity of
measurement and direction of coupling considered, there are at least three different ways in which the strength of coupling can be measured. First, the definitions for import and export coupling are provided when the entity of measurement is the object and the granularity level is the class. Phrases outside and between parentheses capture the situations for import and export coupling, respectively.

- **Dynamic messages.** Within a run-time session, it is possible to count the total number of distinct messages sent from (received by) one object to (from) other objects, within the scope considered. That information is then aggregated for all the objects of each class. Two messages are considered to be the same if their source and target classes, the method invoked in the target class, and the statement from which it is invoked in the source class are the same. The latter condition reflects the fact that a different context of invocation is considered to imply a different message.

- **Distinct method invocations.** A simpler alternative is to count the number of distinct methods invoked by each method in each object (that invokes methods in each object). Note that this is different from simply counting method invocations as we count each distinct method only once. That information is then aggregated for all the objects of each class.

- **Distinct classes.** It is also possible to count only the number of distinct server (client) classes that a method in a given object uses (is used by). That information is then aggregated for all the objects of each class. If we now look at where the calling and called methods are defined and implemented, the entity of measurement is the class and we can provide similar definitions. We then count the number of distinct messages originating from (triggering the executions of) methods in the class, the number of distinct methods invoked by (that invoke) the class methods, and the number of distinct classes from which the class is using methods (that uses its methods).

Table 2.2 shows the formal set definitions of the measures when the granularity is the class, and the scope is the system. An intuitive textual explanation is provided for the first set only: $IC_{OM}(c)$. Other sets can be interpreted in a similar manner.
IC_OM(c): A set containing all tuples (source method, source class, target method, target class) such that there exists an object o instantiating c (whose coupling is being measured) that sends a message to at least one instance of the target class in order to trigger the execution of the target method. The corresponding metric is simply the cardinality of this set. Note that the source class must be different from the target class (c1 ≠ c2), because we are focusing on dependencies that contribute to coupling between classes, not their cohesion. Reflexive method invocations are therefore excluded.

<table>
<thead>
<tr>
<th>Direction</th>
<th>Entity of Measurement</th>
<th>Strength</th>
<th>Set Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Import Coupling</td>
<td>Object</td>
<td>Dynamic messages</td>
<td>IC_OD(c1) = {(m1, c1, l1, m2, c2)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>c1 ∈ c2 ∧ (o, m1, l, o2, m2) ∈ ME}</td>
</tr>
<tr>
<td></td>
<td>Distinct Methods</td>
<td></td>
<td>IC_OM(c1) = {(m1, c1, m2, c2)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>c1 ∈ c2 ∧ (o, m1, l, o2, m2) ∈ ME}</td>
</tr>
<tr>
<td></td>
<td>Distinct Classes</td>
<td></td>
<td>IC_OC(c1) = {(m1, c1, c2)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>c1 ∈ c2 ∧ (o, m1, l, o2, m2) ∈ ME}</td>
</tr>
<tr>
<td>Export Coupling</td>
<td>Object</td>
<td>Dynamic messages</td>
<td>IC_OD(c1) = {(m1, c1, l1, m2, c2)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>c1 ∈ c2 ∧ (o, m1, l, o2, m2) ∈ ME}</td>
</tr>
<tr>
<td></td>
<td>Distinct Methods</td>
<td></td>
<td>IC_OM(c1) = {(m1, c1, m2, c2)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>c1 ∈ c2 ∧ (o, m1, l, o2, m2) ∈ ME}</td>
</tr>
<tr>
<td></td>
<td>Distinct Classes</td>
<td></td>
<td>IC_OC(c1) = {(m1, c1, c2)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>c1 ∈ c2 ∧ (o, m1, l, o2, m2) ∈ ME}</td>
</tr>
<tr>
<td>Class</td>
<td>Dynamic messages</td>
<td></td>
<td>EC_OD(c1) = {(m2, c2, l1, m2, c2)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>c1 ∈ c2 ∧ (o, m1, l, o2, m2) ∈ ME}</td>
</tr>
<tr>
<td></td>
<td>Distinct Methods</td>
<td></td>
<td>EC_OM(c1) = {(m2, c2, m2, c2)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>c1 ∈ c2 ∧ (o, m1, l, o2, m2) ∈ ME}</td>
</tr>
<tr>
<td></td>
<td>Distinct Classes</td>
<td></td>
<td>EC_OC(c1) = {(m2, c2, c2)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>c1 ∈ c2 ∧ (o, m1, l, o2, m2) ∈ ME}</td>
</tr>
</tbody>
</table>

Table 2.2 Summary of Dynamic Coupling Measures (granularity=class, scope=system)
**Higher Granularities**

If one wants to measure dynamic coupling at higher levels of granularity, this can be easily defined by performing the union of the coupling sets of a set of classes or objects, depending on the entity of measurement. For example, if the entity of measurement is the class and the level of granularity is the subsystem, than for each subsystem SS there corresponds a subset of classes that it contains, SC ⊆ 2C, and:

\[ IC_{CM}(SS) = \bigcup (\text{all c} \in SC) IC_{CM}(c) \]

Similarly, when the entity of measurement is the object: For each use case UC there is a corresponding set of participating objects SO ⊆ 2O (that are involved in the UC’s sequence diagram(s)), and:

\[ IC_{CM}(UC) = \bigcup (\text{all o} \in SO) IC_{CM}(o) \]

Similar definitions can be provided for all levels of granularity.

**2.4.2 Dynamic Coupling Measures from early stage of lifecycle**

As explained earlier, a very big difference between object-oriented software development and the traditional way of software development is that OO-software development puts a greater emphasis on the design phase of the software development lifecycle, whereas structured development methods focus more on the implementation phase. That is why object-oriented metrics aimed at the design of an object-oriented system, instead of its code, makes them more suited to the object-oriented paradigm.

Also, if a certain metric provides some kind of insight about the software system under development by using the code of that system as the entity of measurement, but it is possible to derive almost the same measurement, if not the same, using the design if that system as entity of
measurement, then the later one would be more meaningful. This because the implementation phase is at the very last part in an object-oriented software’s lifecycle. Information derived from measures, usually indirect measures, can be very helpful in the early stage of software development, and may be too late at the design phase. Worse, if that information will result in a change of that design, the whole implementation stage is a disaster.

Therefore, in this thesis only measures will be investigated which can be used upon the design of a software system. It that way it is conceivable that dynamic design models (e.g., interaction diagrams in the Unified Modeling Language (UML) [6]) could be used to collect such measures.

For the dynamic measures explained in the previous sub-section, this criterion is against those dynamic coupling measures with ‘dynamic messages’ as the level of strength. These measures require information regarding the statements from where messages are invoked in their source class. However, as explained by Arisholm [3, 4 and 5] that in a UML sequence diagram this would be represented as distinct messages with identical method invocations but different guard conditions, complexity is added to these measures if derived from UML alike models, and according to the PCA analysis (see chapter 6 and 7) these measures are not measuring different data than their variants who differ only is the level of strength. This is the reason why these measures will be excluded in the rest of this thesis.
Chapter 3

Definition of some new Object-Oriented Metrics

In this chapter some additional dynamic coupling measures will be defined, with a new level of strength, class-to-class. This will be followed by the definition of a new way of coupling named ‘neighbor-coupling’. Using this definition of neighbor-coupling, a few dynamic and static neighbor coupling measures will be defined. Finally some properties of these newly defined neighbor-coupling metrics will be explained.

3.1 Some additional dynamic coupling measures

Using the same sets and relations as in the definition of the dynamic coupling measures in chapter 2, a set of new dynamic coupling measures will be defined. Differentiating in the entity of measurement (class or object) and the direction of coupling (import and export) the following table lists four new dynamic coupling measures, with the ‘class-to-class’ level as the level of strength :

Distinct Class-to-Class: Within a run-time session, it is possible to count the total number of distinct class-to-class messages (or method invocations) sent (or received) from one object (or class) to (or from) other objects (or classes), within the scope considered. That information is aggregated for all the objects (or classes). Two messages are considered to be the same if their source and target classes are the same. The four new dynamic coupling measures with the class-to-class level of strength are shown in the table below:
3.2 Introducing Neighbor-Coupling

The need for a new metric

Coupling measures are not new. However, new kinds of coupling measures, mostly dynamic ones, are being defined recently. All of these measure in the same way the other classes to which some class is coupled to, where the definition of 'coupling' just changes, along with the direction, entity of measurement and strength. According to much research done, including this thesis, some of these coupling measures are very effective in predicting useful external attributes of software systems. However, still there are two aspects in object-oriented software development which are not covered by existing measures:

Table 3.1 Summary of new Dynamic Coupling Measures (granularity=class, scope=system)
1. Taking figure 3.1 as an example, class c2 is coupled to class c1, because of the messages sent from class c1 to class c2. Further, class c3 is coupled to class c2 because of the messages sent from class c2 to class c3. However, it may be the case that because of the message m1 sent from class c1 to class c2, that class c2 sends a message \texttt{init} or \texttt{m2} to class c3. This phenomenon is not captured by current coupling measures. This does not state that current coupling measures are incomplete. They just have another purpose/definition/goal.

2. If a software system has two classes (c1 and c2), with the same amount of coupling (e.g. CBOImp), then they are considered in existing (prediction) models to have the same complexity value, or some other external measure which is predicted by mostly the coupling of a class. One of such external measurements is the fault-proneness (chapter 5) for which import coupling tends to be a very significant predictor. However, if the classes to which c1 is coupled have a very low coupling value themselves, if not zero, but all the classes c2 is coupled to have a high coupling value, then this can be, with respect to complexity as an example, interpreted as class c2 operating in a more complex environment than class c1, and therefore being in some way more complex than class 1. This effect is not covered by existing coupling metrics.

![Diagram](image.png)

\textbf{Figure 3.1: Coupling of class c1 does not include class c3}
To solve the problems just explained, a new measure is defined, neighbor coupling. The purpose of neighbor coupling is not to measure a class its coupling, but rather the coupling of a collection of classes to which some class c1 is coupled to, with the rest of the system, which will have a more stable effect. These classes are the direct environment of the class under measurement, class c1 in this example. This coupling value will be assigned as the neighbor coupling of class c1. The collection of classes used here, which make up the direct environment of class c1, are those classes, c1 is coupled to using some existing coupling definition/function like e.g. CBOImp. To picture this new coupling measure, figure 3.2 depicts an example. The red class is the class under measurement. The black classes are the classes this red class is coupled to e.g. thru CBOImp. In the example, the red class its CBOImp value is 5. The blue classes are the classes to which the black classes are coupled to. It is this collection of blue classes which make up the neighbor coupling for the red class, and which has the value of 15. So, another way of looking at Neighbor coupling is the coupling of a class its environment at level 1. It is off course possible to count the (distinct) coupling of the blue classes as the neighbor coupling of the red class, which would be its neighbor coupling at level 2, but this “level attribute” of neighbor coupling is discarded in the rest of the thesis, and considered to have the value 1. A more realistic example of neighbor coupling is given below (Figure 3.3) after the formal definition of neighbor coupling.

Figure 3.2: The idea of Neighbor Coupling measures
3.3 Defining and validating some Neighbor-Coupling measures

Formally Defining (Static) Neighbor Coupling

Using the formal framework depicted by Figure 2.2 of section 2.4.1, which will allow to provide precise and unambiguous definitions, neighbor coupling measures will be defined. Because the dynamic Neighbor Coupling measures are somewhat more complex then the static variants, static Neighbor Coupling will be defined first, followed by an example, after which some dynamic neighbor coupling measures will be introduced. The following sets are defined:

C: Set of classes in the system (depending on the chosen granularity). C can be partitioned into the subsets of application classes (AC), library classes (LC), and framework classes (FC). Some of these subsets may be empty, C=AC U LC U FC and AC \cap LC \cap FC = \emptyset. Distinguishing such subsets may be important for defining the scope of measurement. The mathematical function for C, containing n classes is:

\[ C = \{c_1, c_2, c_3 \ldots c_n\} \]

M: The set of methods in the system defined within the classes part of C.

Rmc: This set refers to the methods being defined in classes, a binary relation from the set of methods to the set of classes:

\[ Rmc \subseteq M \times C \]

Using some coupling function f which takes C,M and Rmc as parameters (and which for static coupling may be CBO, CBOImp, CBOExp or some other coupling function) and whose output is defined by a collection F of the following elements:

\[(c_1,c_2)\]
which stands for \( c_1 \) being coupled to \( c_2 \) (according to \( f \)), a set \( \textbf{D-ENV} \) for each class \( c_k \) is computed, where \( \text{D-ENV}(c_k) \) stands for \( c_k \) its *direct environment*, and consists of all the *distinct* classes to which \( c_k \) is coupled thru the function \( f \):

\[
\textbf{D-ENV}(c_k) = \{ (cp) | (\forall (ck, cp) \in F) \land cp \neq ck \}
\]

Using these definitions, Neighbor Coupling for a class \( c_k \) is defined as the collection of *distinct* classes coupled to at least one of the classes if \( \text{D_ENV}(c_k) \), which them selves may not be part of \( \text{D_ENV}(c_k) \):

\[
\textbf{NG}(c_k) = \{ (cp) | (\forall (co, cp) \in F) \land (co \in \text{D-ENV}(c_k)) \land ! (cp \in \text{D-ENV}(c_k)) \land cp \neq c_k \neq co \}
\]

The table below gives the names for the static neighbor coupling measures with their respective coupling function \( f \). The formulas above are valid for all these three measures.

<table>
<thead>
<tr>
<th>Coupling Function/Measure</th>
<th>Neighbor Coupling Measure</th>
</tr>
</thead>
<tbody>
<tr>
<td>CBO</td>
<td>( \text{NG}_\text{CBO} )</td>
</tr>
<tr>
<td>CBOImp</td>
<td>( \text{NG}_\text{CBOImp} )</td>
</tr>
<tr>
<td>CBOExp</td>
<td>( \text{NG}_\text{CBOExp} )</td>
</tr>
</tbody>
</table>

*Table 3.2: Naming conventions for static neighbor coupling measures used in this thesis*

*Working Example*

As an example, a system will be used, whose classes and coupling relations (where the function \( f \) may represent e.g. CBOImp) are depicted in figure 3.3. The values of most the previously defined sets become:

\[
C = \{ c_1, c_2, c_3, c_4 \ldots c_{22} \}
\]
F = \{(c1, c12), (c1, c9), (c1, c6), (c1, c3), (c1, c19), (c1, c14), (c9, c11), (c9, c8), (c9, c7), (c9, c4), (c3, c4), (c3, c5), (c3, c20), (c2, c5), (c2, c20), (c2, c21), (c2, c22), (c14, c13), (c14, c15), (c14, c16), (c16, c17), (c17, c16), (c19, c16), (c19, c18)\}

**Figure 3.3: Example of Neighbor Coupling**

Focusing on class \(c1\) in this example:

\[D-ENV(c1) = \{ c2, c3, c6, c9, c12, c14, c19\}\]

If the coupling used in this picture represents CBOImp:

\[NG_{CBOImp}(c1) = \{c4, c5, c7, c8, c11, c13, c15, c16, c18, c20, c21, c22\}\]

*Validation of Neighbor Coupling Measures*

As a theoretical validation of a metric, one has to show that that metric has intuitive properties. The validation of software metrics however is still an unclear part within computer science. Some researchers have proposed some properties which must hold for all metrics (in order to validate them), while others criticize these. Also, there is an implicit assumption in the software engineering community that validation of a measure is not sufficient. Specifically, it is expected that ‘validation’ must also entail the demonstration that the measure is itself part of a valid
prediction system [25]. In the next part, Neighbor coupling will be inspected for its intuitive properties.

As an example coupling function, CBOImp will be used, but the same demonstration is valid for other coupling measures which consist of distinct elements of the following format:

$$(c_1, c_2),$$

Where $c_1$ and $c_2$ represent two different classes, and is interpreted as class $c_1$ being coupled to class $c_2$.

Metric properties:

Non-negativity
It is not possible for the Neighbor Coupling measures to be negative because they measure the sum of the cardinality of sets, e.g., for each coupling function used, the cardinality is non-negative. The sum of the cardinality of a collection of classes is therefore also non-negative.

Null values
At the System level of granularity, if non-system classes are filtered out, the Neighbor coupling of the system is always zero, since messages between the system and non-system classes (e.g. library classes) are discarded. This means that the direct environment of the System is empty. At the class level, the Neighbor Coupling of a class $c_1$ is null, only if the coupling of that class is null or if the direct environment of that class is not coupled to the rest of the system (it is possible that library classes are not considered). When as the coupling function a coupling measure is used which accounts for import and export coupling (like CBO), this means that there is no interaction between the direct environment of $c_1$ and other classes. This is only possible if the direct environment of $c_1$ extended with $c_1$ contains the whole system.

Monotonicity
If a class $c$ is modified such that at least one instance $o$ sends/receives more messages, its import/export coupling can only increase or stay the same [3, 4, and 5]. When the coupling of a class $c_1$ increases, its direct environment can only be extended with zero, one or more classes, which results in the fact that the neighbor coupling of class $c_1$ can only increase or stay the same.
If the neighbor coupling of class c1 stays the same then this means that the extra coupling could no extend the direct environment with another distinct class, or the additional classes added to its direct environments had no coupling, or the classes they were coupled to were already part of class c1 its neighbor coupling, or equal to class c1. Because the Neighbor Coupling cannot decrease by such a change, this complies with the monotonicity property.

**Impact of merging uncoupled classes**

Assuming c' is the result of merging c1 and c2, thus transforming system S into S', for any Coupling measure the following properties hold at the class and system levels [3, 4 and 5]:

Coupling (c1) + Coupling (c2) ≥ Coupling (c')

Coupling(S) ≥ Coupling(S')

The neighbor coupling of c’ is only equal to that of c1’s, if the neighbor coupling of classes c1 and c2 are equal, thus containing the same classes or if the neighbor coupling of c2 is zero. If the neighbor coupling of c1 and c2 are not zero, and not totally equal, then the following will hold:

|NG (c1)| ≤|NG (c')| & |NG (c2)| ≤|NG (c')|

This, because NG(c1) and NG(c2) were not equal. So there are some additional elements (classes) which are only part of c1 or c2, but not both. When merging both collections, the new collection will be bigger than both NG(c1) and NG(c2). This leads to NG(c’) being lower or equal to the sum of NG(c1) and NG(c2) in the general case:

|NG (c1)| + |NG (c2)| ≥|NG (c')|

At the System level of granularity, if non-system classes are filtered out, the Neighbor coupling of the system is always zero, since messages between the system and non-system classes (e.g. library classes) are discarded.

**Merging coupled classes**

Same holds as when merging uncoupled classes, adding the following possibility: class c1 and c2 are coupled, and both their neighbor coupling consists of exactly the same elements (classes). C2 is part of c1’s neighbor coupling and the other way around. When merging c1 and c2 to c, the
neighbor coupling of \( c \) will contain all the elements of the neighbor coupling of \( c_1 \) without element \( c_2 \), which is the same as all the elements of \( c_2 \)’s neighbor coupling without element \( c_1 \). So in this scenario, the neighbor coupling of \( c \) is less than the sum of the neighbor coupling of \( c_1 \) and \( c_2 \).

Based on the property analysis above, we can see that neighbor coupling measures seem to exhibit intuitive properties with a few exceptions. A property which neighbor coupling does not hold compared to coupling measures is that for a coupling measure its total count for import coupling is equal to the total count of export coupling. The symmetry property is intuitive, because anything imported by a class or object has to be exported by another class or object, respectively. For neighbor coupling this is not the case. Another property of neighbor coupling is that the neighbor coupling of a class \( c_1 \) can be changed by changing one of the classes \( c_1 \) is coupled to. So without changing class \( c_1 \), its neighbor coupling can be changed. Another exception is that there exists a possibility that when merging two coupled classes, the neighbor coupling of the newly formed class might be less than the sum of the neighbor coupling of the merged classes. Also, as will be seen below when defining dynamic neighbor coupling metrics, the granularity of neighbor coupling has to be the class level or higher. This because the direct environment of a class consists of classes at the minimum level of granularity. So therefore, the used coupling function for determining a class its direct environment has to have at least the class level of granularity.

As mentioned earlier, metric validation is still a discussion point within computer science. This makes it not easy to conclude whether a metrics is valid or not. In this thesis some aspects of neighbor coupling have just been explained. These have been used to validate coupling metrics [3, 4 and 5]. Because neighbor coupling is a different metric than coupling metrics, these aspects are not being interpreted as a validation, but rather to make the inner workings of neighbor coupling clear to the reader. Another general validation used for metrics is to proof them being good predictors of some external attribute. In this thesis neighbor coupling metrics will be tested for their predictive power of the fault-proneness of two software systems (chapter 7).

*Dynamic Neighbor Coupling Measures*
The function $f$ can be substituted by any dynamic coupling measure whose elements contain only the from-class and to-class or from-object and the to-object of messages or method invocations. This means that the dynamic coupling measures in the previous chapter are not being used for neighbor coupling. Instead, the four dynamic coupling measures defined in this chapter will be used and named in the following way:

<table>
<thead>
<tr>
<th>Coupling Function/Measure</th>
<th>Neighbor Coupling Measure</th>
</tr>
</thead>
<tbody>
<tr>
<td>IC_OCC</td>
<td>NG_IC_OCC</td>
</tr>
<tr>
<td>IC_CCC</td>
<td>NG_IC_CCC</td>
</tr>
<tr>
<td>EC_OCC</td>
<td>NG_EC_OCC</td>
</tr>
<tr>
<td>EC_CCC</td>
<td>NG_EC_CCC</td>
</tr>
</tbody>
</table>

Table 3.3: Naming conventions for dynamic neighbor coupling measures used in this thesis

The same validation for the static neighbor coupling measures hold for these dynamic measures.

### 3.4 The yet missing indicator of usage of Object-Oriented features: DII

In much research done within the area of dynamic software metrics, the resulting measures of two or more object-oriented software systems are analyzed in order to draw some conclusions. In some of these studies the difference between dynamic measures and their static variants are being studied, having as a hypothesis that dynamic metrics are more precise than their static variants and thus can be more helpful in building prediction models to predict some external attribute. Why dynamic metrics can be more precise than their static variant is because at run time different messages can be triggered than would suspected from static code analysis. See the example in section 2.3.3.

Something that was missing when reading these papers/articles is that never the level of usage of object-oriented features is regarded (2.3.1). One may build a software system using an object-oriented language like e.g. Java without using any object-oriented features like inheritance. In that case dynamic measures will not really be any more specific than their static variants and the previous hypothesis will not be valid. So in order to derive justified conclusions when comparing analysis results regarding dynamic and static measures between two or more software systems, some metrics which measures the level of usage of object-oriented features should be taken into
account. Because this was not the goal of this project, just the initial steps were taken towards such a metric, which is inevitable for deriving justified conclusions regarding dynamic metrics. As will be seen at a later stage of this thesis, the metric defined below will be useful in explaining certain artifacts regarding data analysis upon dynamic and static measures of two object-oriented software systems.

At run-time, each message has the following elements: (o1, m1, o2, m2), which stands for a message being send from object o1’s method m1 to method m2 of object o2. From this, information is available regarding 2 method invocations:

1. Invocation of method m1 upon object o1.
2. Invocation of method m2 upon object o2.

If object o1 is of type class c1, and class c1 does not implement method m1, but inherits from any other class c3, then polymorphism is used, even if method m1 is declared in a super class of c3. The same holds for method m2 and object o2 (which is of type c2 for instance). For simplicity sake the possibility that m1 is implemented in class c1, but is declared in one if c1’s superclasses, which should also account for usage of polymorphism (specialization), is not accounted for. This along with other object-oriented information like inheritance and dynamic binding could be investigated/included in the future.

The distinct occurrences when a message is send from/to a method m upon object o of type class c, where m is not implemented or declared in class c, are accounted for and referred to as the distinct occurrences of invocation thru inheritance. This number can be used as an indication of usage of object-oriented features. However, in order to compare two or more different software systems, this would not be valid because the systems could also differ in size. And if one systems is bigger than the other one, it should not necessary mean that also more usage is made of object-oriented features. Therefore, this derived number should be divided by the number of distinct invocations of that system, which will eliminate the size effect. As we will see later on, it is also handy to multiply this number with 100, because the number of distinct invocations is always higher than the accounted occurrences.
Because this metric is used to indicate the usage of object-oriented features, which may play a role at run-time, this metric is called *dynamic invocation indicator* (DII) and is computed as follows:

$$DII = \left( \frac{\text{Distinct occurrences of invocation thru inheritance}}{\text{Distinct Invocations}} \right) \times 100$$

In section 7.6 this metric will be used to explain the differences noticed when analyzing the collected measures for two selected software systems.

**Chapter 4**

**Measuring Dynamic Coupling in Java**

In this chapter the Java programming language will be introduced and explained. This information will be needed when the Java programming language will be used to trace running java programs using the Java Debugging Interface (JDI) in order to collect run-time data of these programs. This collected data will be used further to compute dynamic coupling measures as explained/defined earlier. Also, the Bloat library will be used to examine Java class files, in order to extract static information, from which static measurements will be computed.

**4.1 Java and the Java Virtual Machine**

**4.1.1 The java programming Language**

A new object-oriented programming language called Java was introduced by Sun Microsystems in 1995 which was close to C/C++. Its features of portability, robustness, simplicity and security have made it increasingly popular within the software community [6, 30 and 40]. Java combines a wide range of language features found in different programming languages, for example, an object-oriented model, multithreading, exception handling and automatic garbage collection. A disadvantage is that these features come at the expense of a decrease in performance.

In recent years the Java programming language has integrated itself into our every day lives. Java technology is involved in the creation of devices such as televisions, VCR’s, audio components,
fax machines, scanners, printers, cell phones, smart personal digital assistants, pagers, keys to homes and cars, watches or smart cards.

### 4.1.2 The Java Virtual machine & its components

At the heart of Java technology lies the Java Virtual machine (JVM). This is the abstract computer on which all Java programs run and with which Java class files, Java’s Application Programming Interface (API), and the Java language work together to make the Java phenomena possible. The ability to implement the JVM on various platforms is what makes Java portable.

In traditional languages, such as C/C++, Pascal or FORTRAN, the source code is directly compiled to the instruction set of the target Operating Systems Central Processing Unit (OS-CPU). In contrast Java’s compiled code, known as byte code, is platform independent. The JVM is the interface between compiled Java programs and any target hardware platform. It is also this component of Java technology responsible for the small size of its compiled code, and Java’s ability to protect users from malicious programs.

### 4.1.3 Execution of a java program

A Java program is a collection of class definitions written in the Java language. A Java compiler then compiles or translates this into a collection of bytes that are represented in a form known as the class file format. This is a platform independent intermediate representation of the program. As an alternative to being stored in a file these bytes can also be kept in a database, across a network or as part of a Java Archive File (JAR).

The class file contains byte code instructions for the JVM. This file will have the same semantics as the original Java source code. The JVM will execute these instructions to produce executable code. Figure 4.1 illustrates the steps involved in the execution of a Java program. As long as the original semantics of the instructions are obeyed, the JVM is free to perform the actions specified
by the byte code in any way it sees fit. There are a number of implementation techniques a JVM has to choose from and the memory layout is also determined by the JVM implementation. The byte code instructions can be interpreted or alternatively the JVM can translate them into native machine code.

![Figure 4.1: Steps in the execution of a Java program](image)

### 4.2 Java Debugger Interface and Bloat

#### Java Debugger Interface

The **Java Platform Debugger Architecture (JPDA)** consists of three interfaces designed for use by debuggers in development environments for desktop systems. The **Java Virtual Machine Debugger Interface** defines the services a VM must provide for debugging. The **Java Debug Wire Protocol** defines the format of information and requests transferred between the process being debugged and the debugger front end, which implements the **Java Debug Interface**. The Java Debug Interface defines information and requests at the user code level.

The Java Debug/Debugger Interface (JDI) is a 100% Java implemented interface, which defines debugging information and request at a user code level. This interface greatly facilitates the integration of debugging capabilities into development environments, which makes it possible to extract selected run-time information regarding a Java program. The JDI library will be used by a newly developed tool, Jrev (see below).

#### Bloat
BLOAT [46] is Java bytecode optimizer written entirely in Java. By optimizing Java bytecode, code improvements can occur regardless of the compiler that compiled the bytecode or the virtual machine on which the bytecode is run. BLOAT performs many traditional program optimizations such as constant/copy propagation, constant folding and algebraic simplification, dead code elimination, and peephole optimizations. Additionally, it performs partial redundancy elimination of arithmetic and field access paths.

In this project (Jrev), the Bloat library was used to extract information regarding the classes within a Java program, the methods defined within these classes, their attributes, their superclasses, method invocations (static) etc. This collected data is used further to derive/compute static metrics (CBO, CBOImp, CBOExp, NAImp, NMImp and some static neighbor coupling metrics) regarding the input java program.

The Bloat library and JDI are used by Jrev in order to output certain dynamic and static coupling metrics. Also some size metrics are provided and are used all together in the next stage of this project (chapter 5 and higher) where their relation towards the fault-proneness of software classes is studied. In the next section the Jrev tool will be introduced.

4.3 Jrev: tracing java program execution

Jrev is a tool which extract from a running Java program the messages send between the objects. It uses this data collection to derive/compute all the dynamic metrics covered in this thesis. Jrev also consists of a static part, which extracts from Java class files information regarding the methods and attributes declared within that class, its super classes ect., using the Bloat library. Using this information, all static measures covered in this thesis are computed, including the size measures NMImp and NAImp. Jrev is used to derive all these measures from Velocity and Tomcat. In chapter 7 these measures will be used further for data analysis.

In appendix A, the design of a Jrev is given. Appendix B provides the public interface (API) of Jrev for usage as a service by other Java applications.
4.4 Velocity

In this section, The Velocity software will be explained. It is used in this project among Tomcat to derive the considered metrics and analyze their relation with the fault-proneness of their classes.

Velocity [19 and 29] is a Java-based template engine. It permits anyone to use a simple yet powerful template language to reference objects defined in Java code.

When Velocity is used for web development, Web designers can work in parallel with Java programmers to develop web sites according to the Model-View-Controller (MVC) model, meaning that web page designers can focus solely on creating a site that looks good, and programmers can focus solely on writing top-notch code. Velocity separates Java code from the web pages, making the web site more maintainable over its lifespan and providing a viable alternative to Java Server Pages or PHP.

Velocity's capabilities reach well beyond the realm of the web; for example, it can be used to generate SQL, PostScript and XML from templates. It can be used either as a standalone utility for generating source code and reports, or as an integrated component of other systems. For instance, Velocity provides template services for the Turbine web application framework, together resulting in a view engine facilitating development of web applications according to a true MVC model. For more information, see also the online guide at Jakarta Velocity.

In order to trace and derive dynamic coupling measures for Velocity, a test program was created, called “test.Vel12TestSuite”. This program initiates Velocity, defines a context, executes some commands in order to extend coverage of Velocity and finally runs all the test cases supplied with Velocity.

This test program is given to Jrev as input and during execution it is traced. All the measures are written to file with the corresponding class name to make a distinction between measures belonging to the same class. The Velocity jar file is given to Jrev to compute all the static
measures in the same order as the dynamic part, by using the “-order” option and supplying the output file of all the classes traced earlier. All the metrics are combined with a small script, which produces a text file compatible with Mathlab. In this text file, on each row, all the metrics are written for a certain class of Velocity. This file will be used in chapter 6 and 7 using Mathlab, for further analysis.

4.5 Tomcat

Apache Tomcat [16 and 18] is a web container developed as the Apache Software Foundation. Tomcat implements the Servlet and the Java Server Pages (JSP) specification from Sun Microsystems, providing an environment for Java code to run in cooperation with a web server. It adds tools for configuration and management but can also be configured by editing configuration files that are normally XML-formatted. Because Tomcat includes its own HTTP server internally, it is also considered as a standalone web server. In this thesis tomcat 4.0 was used, because of the available fault data. Tomcat 4 implements Servlet specification 2.3 and JSP specification 1.2.

For the same purpose as Velocity, Tomcat is used to extract all the needed measures for further analysis. For collecting dynamic data using Jrev, Tomcat was started as a stand alone application within Jrev, see figure 4.2, in order for Jrev to collect run-time information (messages between objects ect.). While Jrev was tracing Tomcat, the Tomcat Test suite was executed as another application, which sends different request to the server (running within Jrev) in order to extend coverage of the executed Tomcat code. Also, using a web-browser, functionality of Tomcat was excised.
In this section, the underlying base of the Analysis/Study regarding the relation between the mentioned dynamic coupling metrics and fault-proneness of an object-oriented class will be explained. First I make the distinction between dependent variables and independent variables, all measures used in this thesis will be allocated to one of these two kinds, and the last part of this chapter encapsulates the entire hypothesis of this project, which will be tested in the next chapters.

The goal of this study is to empirically investigate the relationships between the object-oriented dynamic coupling design measures discussed earlier (at the class and object level of measurement), the static and dynamic neighbor coupling measures, as defined earlier, and the fault-proneness of a software system/part. According to chapter 2, these static and dynamic (neighbor) coupling measures are internal metrics, which measure attributes of the “Product” entity, because they use the design of a system, or its code (Jrev) in order to compute/derive the measures. The fault-proneness of a system is considered an external attribute, simply because such a measure cannot be measured directly. That is why the goal of this thesis is to measure this attribute indirectly by using a prediction model (see section 2.2 for the definition of a model).
5.1 Dependent Variable

To investigate the relation between some metrics and the fault-proneness of a system, we need to select a suitable and practical measure of fault-proneness as the dependent variable for our study. In this thesis, fault-proneness is defined as the probability of detecting a fault in a class. As described in the chapter below, a much used classification technique will be investigated, called logistic regression, which is based on predicting event probabilities. In our instance, an event is a detection of a fault in a class due to a failure reported by a user or a discovered fault in the software system/component under investigation. Usually, the probability of fault detection, or any other external quality attributes, is described as a function of the structural properties of the classes. This analysis is further explained in the next chapter. Clearly, other choices for the dependent variable could have been used (e.g., fault density) but, to a large extent, such a definition is driven by the choice of the modeling technique used for data analysis. Furthermore, the alternative choice of fault density as a dependent variable has its own problems. Even when there is no causal relationship between size and number of faults, a negative correlation between size and fault density can be found [10, 15]. This is, however, a pure mathematical artifact, which makes any analysis of the impact of size on fault-proneness difficult.

5.2 Independent Variables

The measures of coupling, cohesion, inheritance and size identified in a literature survey on object-oriented design measures [15] are possible independent variables to use in such studies, where one wants to investigate the relationship between structural properties of a design and its external quality attributes. In this study, however, we specifically focus on dynamic coupling. We focus on those dynamic coupling metrics which are design measurements since we want the measurement-based models investigated in this thesis to be usable at early stages of software development. In a later chapter we will work out an example in making such a model useful in early stages of software development, by altering an existing software life cycle. Furthermore, we use measures defined at the class and object level of measurement since we want to incorporate the effect of object-oriented features like inheritance, polymorphism and dynamic binding, as explained in earlier chapters.
A total of 8 dynamic coupling measures defined by Arisholm [3, 4 and 5] are considered for analysis. These are IC_OM, IC_OC, IC_CM, IC_CC, EC_OM, EC_OC, EC_CM and EC_CC. The **_D dynamic coupling measures are not considered because they include information regarding the lines of code from where methods are invoked. This is contrary to our statement to investigate a fault-proneness prediction model usable in early stages of software development.

The additional dynamic coupling measures defined earlier, IC_OCC, IC_CCC, EC_OCC and EC_CCC, will also be used in this project, along with their dynamic neighbor coupling measures NG_IC_OCC, NG_IC_CCC, NG_EC_OCC and NG_EC_CCC.

To compare the results of the study with respect to these independent variables, I also include some static coupling measures, CBO, CBOImp and CBOExp, which have been proven in previous studies to be good predictors of the fault-proneness of software classes. Also, for investigation of neighbor coupling, their neighbor coupling variant (NG_CBO, NG_CBOImp and NG_CBOExp respectively) will also be included.

Another important aspect to consider is the size of classes. We will investigate the relationship of these coupling metrics to size measures NAImp and NMImp, and will investigate whether all the former, more complex measures help build better predictive models than size alone. As mentioned earlier, the size of designs (e.g., class designs) is a necessary part of any predictive model [15]. This is mostly justified by the fact that the size of any software artifact determines, to some extent, many of its external properties such as fault-proneness or effort. On the one hand, we want our predictive models to account for size. But in many cases, e.g., in the case of fault-proneness models, and for practical reasons, we need them to capture more than size effects. A model that systematically identifies bigger classes as more fault-prone would a priori be less useful: the predicted fault-prone classes are likely to cover a larger part of the system; the model thus could not help to focus inspection and testing efforts very well [12]. For the purpose of assessing the usefulness and practicality of all the predictive models we obtain, we will compare the increase in potential fault detection rate and the increase in size to be tested or inspected. Also, there will be a comparison between size-only models and models including more complex measures, to determine if the extra effort for these more complex measures is worthy or not.
5.3 Hypotheses

In this thesis, a number of hypotheses will be tested which relate various design measures (e.g. dynamic coupling metrics) to fault-proneness. Our hypotheses are mostly derived from the causal chain depicted in Figure 5.1: The structural properties of a class, measured by the coupling, cohesion, and inheritance measures, affect the cognitive complexity of the class [15]. By cognitive complexity we mean the mental burden of the persons who have to deal with the class (developers, inspectors, testers, maintainers, etc.). It is believed that it is the, sometimes necessary, high cognitive complexity of a class which causes it to display undesirable external qualities, such as increased fault-proneness, or decreased maintainability and testability. The external class quality attributes are therefore indicators of the cognitive complexity.

![Figure 5.1: Relationships between structural class properties, cognitive complexity and external quality attributes.](image)

The structural properties of a class are not, however, the only factors which affect a class’s cognitive complexity. For instance, the completeness and traceability of the documentation will also have an impact. Cognitive complexity is also dependent on the individuals who deal with the class and is to some degree subjective. Even if we restrict our attention to structural properties, their relationship to external quality attributes will be affected (i.e., interactions) by other factors such as the software engineer’s capability and experience and the development methodology in place. So, although an attempt is made here to identify general trends, we have to expect variability, whose extent is not known. In the context of the more general program of research depicted in Figure 5.1, we focus, in this study, on an external class quality which can be measured easily and objectively: fault-proneness.

The following hypotheses will be tested in this study:

- **H-IC** (for all import coupling measures): A class with high import coupling is more likely to be fault-prone than a class with low import coupling. A class with high import
coupling relies on many externally provided services. Understanding such a class requires knowledge of all these services. The more external services a class relies on, the larger the likelihood to misunderstand or misuse some of these services. Therefore, that class is more difficult to understand and develop, and thus likely to be more fault-prone. Dependent variables on which this hypothesis lies are IC_OM, IC_OC, IC_OCC, IC_CM, IC_CC, IC_CCC, CBO and CBOImp.

- **H-EC** (for export coupling measures): A class with high export coupling is more likely to be fault-prone than a class with low export coupling. A class with high export coupling has a large influence on the system: many other classes rely on it. Failures occurring in a system are therefore more likely to be traced back to a fault in the class, i.e., the class is more fault-prone. Dependent variables on which this hypothesis lies are EC_OM, EC_OC, EC_OCC, EC_CM, EC_CC, EC_CCC, CBO and CBOExp.

- **H-IC-ENV** (for all import neighbor coupling measures): A class’s direct environment, which is defined by the collection of classes to which this subject-class is coupled to thru some import coupling measure (dynamic or static), with high import coupling with the rest of the software system/part, suspects that that class is more likely to be fault-prone when its direct environment has low import coupling with the rest of the software system/part. A class’s direct environment with high import coupling relies on many externally provided services. Understanding such a collection of classes requires knowledge of all these services. The more external services these collection of classes rely on, the larger the likelihood to misunderstand or misuse some of these services. Dependent variables on which this hypothesis lies are NG_IC_OCC, NG_IC_CCC, NG_CBO and NG_CBOImp.

- **H-EC-ENV** (for all export neighbor coupling measures): A class’s direct environment, which is defined by the collection of classes to which this subject-class is coupled to thru some export coupling measure (dynamic or static), with high export coupling with the rest of the software system/part, suspects that that class is more likely to be fault-prone when its direct environment has low export coupling with the rest of the software system/part. A class’s direct environment with high export coupling has a large influence on the system: many other classes rely on it. Failures occurring in a system are therefore more likely to be traced back to a fault in the class, i.e., the class is more fault-prone.
Dependent variables on which this hypothesis lies are NG_EC_OCC, NG_EC_CCC, NG_CBO and NG_CBOExp.

- **H-DC**: Because of object-oriented features like inheritance, polymorphism and dynamic binding, the resulting static coupling measures on such object-oriented software systems are imprecise as they do not perfectly reflect the actual coupling taking place among classes at run-time. Therefore, dynamic coupling measures are being considered as more precise measures than static variants, and because they represent the software system’s properties much more precise, dynamic coupling measures are considered to be better predictive measures of the fault-proneness of a class than static coupling measures.

These hypotheses will be evaluated in section 7.7 after completing the analysis on Velocity and Tomcat data using the framework of chapter 6.
Chapter 6

Data Analysis Approach

In this chapter, the data analysis framework carried out in the next chapter will be explained. Mostly the empirical study design of Briand [11, 13, 15 and 24] is used. All these statistic techniques which are not already part of Matlab were implemented for analyzing the previous collected measures data taken from Velocity and Tomcat. The outcome of this analysis is covered in chapter 7.

6.1 Descriptive Statistics

For descriptive statistics, the distribution (mean, median, and interquartile ranges) and the variance (standard deviation) give a good overview of the data. Low variance measures do not differentiate classes very well and therefore are not likely to be useful predictors. The range and distribution of a measure determines the applicability of subsequent regression analysis techniques [14]. According to Briand [15], these statistics allow researchers to determine if the data collected across different studies stem from similar populations. If not, this information will likely be helpful to explain different findings across studies. For more information regarding descriptive statistics and mathematics see [14, 15, 28 and 45].
Mean
If $X_1, X_2, X_3, \ldots, X_n$ are a set of $n$ values of a measurement, then the arithmetic mean (or simply mean) is the sum of all these $n$ values, divided by $n$:

$$\bar{X} = \frac{\sum_{i=1}^{n} X_i}{n}$$

Median
If the values of the measurement are arranged in ascending order of magnitude, the Median is the middle element of the values if the number of measurements is odd or the mean of the two middle items of the ordered collection if the number of measurements is even.

Quartiles
When the values of the measurement are arranged in ascending order of magnitude, the lower quartile ($Q_1$) is the mid-way element of these ordered values, between the lower extreme and the Median, and the upper quartile ($Q_3$) is the mid-way element between the Median and the upper extreme.

Standard Deviation
Unfortunately, the Mean of a data collection does not tell us a lot about the data except for a sort of middle point [45]. For example, these two data sets have exactly the same Mean, but are obviously quite different:

$[0 \ 8 \ 12 \ 20]$ and $[8 \ 9 \ 11 \ 12]$

The difference lies in the spread of the data. The Standard Deviation (SD) of a data set is a measure of how the data is spread. The mathematical formula for computing the SD is:
6.2 Outlier Analysis

In statistics, an outlier is a single observation "far away" from the rest of the data. In other words, outliers are data points which are located in an empty part of the sample space [14, 15]. In most samplings of data, some data points will be further away from their expected values than what is deemed reasonable. This can be due to systematic error or faults in the theory that generated the expected values. Outlier points can therefore indicate faulty data, erroneous procedures, or areas where a certain theory might not be valid. Inclusion or exclusion of outliers can therefore have a large influence on the analysis results and prediction models. It is important that conclusions drawn are not solely dependent on a few outlying observations, otherwise the statistical results and its interpretation are unstable and cannot be reliably used. According to Briand [15], it is particularly crucial to ensure that differences in observed trends are not due to singular, outlying data points. For these reasons it is necessary to identify outliers, test their influence, and possibly remove them to obtain stable results. Below univariate and multivariate outliers will be discussed.

In descriptive statistics, a box plot (also known as a box-and-whisker diagram) is a convenient way of graphically depicting the five-number summary, which consists of the smallest observation, lower quartile, median, upper quartile and largest observation. Below examples are given of Univariate and Multivariate outliers using the box plot.

6.2.1 Univariate outlier detection

When the data sample consists of only one dimension, it is called a univariate sample. Its outliers are therefore called univariate outliers. Figure 6.1 gives an example of univariate outliers. It is easy to see that above 500000, the data values are far from the mean and most of the data.
There exist different techniques to identify outliers. Manual inspection of scatter plots is the most common approach to outlier detection. Manual detection of outliers suffers from the two basic limitations of data visualization methods: subjectiveness and poor scalability [37 and 38]. The analysts have to apply their own subjective perception to determine the parameters like “very far away” and “low frequency”. An objective, quantitative approach to unsupervised detection of numeric outliers is described in [41]. It is based on the graphical technique of constructing a box plot, which represents the median of all the observations and two hinges, or medians of each half of the data set. Most values are expected in the interquartile range (IQR) located between the two hinges. Values can be identified as mild or extreme outliers:

Defining Q1 and Q3 to be the first and the third quartiles, and IQR to be the interquartile range: (Q3 – Q1), mild and extreme outliers are defined in the following way:

**Mild outliers**

For a value x, x is identified as a mild outlier if

\[ x < Q1 - 1.5 \times IQR \]

or

\[ x > Q3 + 1.5 \times IQR \]

**Extreme outliers**

For a value x, x is identified as a extreme outlier if

\[ x < Q1 - 3 \times IQR \]

or

\[ x > Q3 + 3 \times IQR \]

Still there is a chance that an as outlier identified value is just a valid value. There exist more complex algorithms to identify univariate outliers [38], but in the rest of this thesis only extreme outliers will be considered for univariate data.
The influence of an identified data point is tested: an outlier is influential, if the significance of the measurements with respect to the hypothesis of the study depends on the absence or presence of the outlier. Such outliers are not considered in the rest of the study [14]. Other possible techniques to test if an outlier is influential are Chauvenet's criterion and Peirce's criterion.

![Box Plots for column X1: POP91](image)

**Figure 6.1: Example of univariate outliers**

### 6.2.2 Multivariate outlier detection

When the data sample consists of more than one dimension, it is called a multivariate sample. Its outliers are therefore called multivariate outliers [9, 26, 27 and 32]. Figure 6.2 gives an example of multivariate outliers.

In our case we examine the relation between a set of independent variables and a dependent variable. The n independent variables span an n-dimensional sample space. To identify multivariate outliers, for each measured data-point, the Mahalanobis Jackknife distance from the sample space centroid is calculated [15]. The Mahalanobis distance is a measure that takes correlation between measures into account. Multivariate outliers are data-points with a large distance from the sample space centroid. A multivariate outlier is over-influential when the significance of the measurements with respect to the hypothesis of the study depends on the absence or presence of the multivariate outlier, and therefore will be removed for further analysis.
The Mahalanobis distance [9, 26, 27 and 32]

Statistical methods for multivariate outlier detection often indicate those observations that are located relatively far from the center of the data distribution. Several distance measures can be implemented for such a task. The Mahalanobis distance is a well-known criterion which uses robust estimators of the covariance matrix and the mean vector. The shape and size of multivariate data are quantified by the covariance matrix. Formally, the Mahalanobis distance from a group of values with mean $\mu = (\mu_1, \mu_2, \mu_3, \ldots, \mu_p)$ and covariance matrix $\Sigma$ for a multivariate vector $x = (x_1, x_2, x_3, \ldots, x_p)$, where each $x_i$ is a collection of observations of the same dimension/variable and $\mu$ is its mean, is defined as:

$$D_M(x) = \sqrt{(x - \mu)^T \Sigma^{-1} (x - \mu)}.$$

For multivariate normally distributed data the values are approximately chi-square distributed with $p$ degrees of freedom ($X^2_p$). Multivariate outliers can now simply be defined as observations having a large (squared) Mahalanobis distance. For this purpose, a quartile of the chi-squared distribution (e.g., the 97.5% quartile) could be considered.

As a point estimate, this statistic is known to evidence small sample bias [36]. A possible approach to the reduction this bias is the Jackknife estimator [32]:

$$D_{2j} = (n)D_m - (n-1)D_m'.$$
Where $D_m'$ is the mean value of $D_m$'s obtained from successively removing each observation (i.e., the mean of the leave-one-out values of $D_m$). This results in $n$ $D_m$'s (where $n$ is the total number of observation), of which $D_m'$ is their mean.

For identifying multivariate outliers, the Mahalanobis Jackknife distance will be computed for each data point. Observations have a large distance, and are considered outliers, if they exceed the 97.5% quartile of the chi-squared distribution.

6.3 Principal Components Analysis and Varimax Rotation

**Principal component analysis** is a way of identifying patterns in data, and expressing the data in such a way as to highlight their similarities and differences. Since patterns in data can be hard to find in data of high dimensions, where the luxury of graphical representation is not available (for dimensions above 2), PCA is a powerful tool for analyzing data [45].

**Principal component analysis (PCA)**, more formally, is a linear transformation that chooses a new coordinate system for the data set such that the greatest variance by any projection of the data set comes to lie on the first axis (then called the first principal component), the second greatest variance on the second axis, and so on. These are the different dimensions the data represent. The latter components represent the data in a lesser way than the first components. The first PC is the linear combination of all standardized variables that explain a maximum amount of variance in the data set. The second and subsequent PCs are linear combinations of all standardized variables, where each new PC is orthogonal to all previously calculated PCs and captures a maximum variance under these conditions [15]. Usually, only a subset of all variables shows large weights and therefore contributes significantly to the variance of each component. To better identify these variables, the loadings of the variables in a given component can be considered. The loading of a variable is its correlation with the component. The variables with high loadings help identify the dimension the component is capturing but this usually requires some degree of interpretation.
If a group of variables in a data set are strongly correlated, these variables are likely to measure the **same underlying dimension** (the same component) of the object to be measured. Therefore, principal component analysis (PCA) is a standard technique to identify the underlying, orthogonal dimensions that explain relations between the variables in the data set. This technique can be very handy, because it is common to see groups of variables in a data set that are strongly correlated.

The other main advantage of PCA is that you can compress the data. This can be done by reducing dimensionality in a dataset while retaining those characteristics of the data set that contribute most of its variance by eliminating the later principal components (by a more or less heuristics decision). These characteristics may be the most important, but this is not necessarily the case, depending on the application. See [45] for a step-by-step procedure of computing the principal components of a data set.

In Software Metrics, PCA can be used to analyze data of multiple dimensions and deriving the important components. For each such component, it is possible to derive which variables have a high correlation, by analyzing the important variables with respect to these components individually. For instance, if the data consists of 5 dimensions and dimensions 2 and 5 have a high (where great needs to be defined eg. more than 60%) contribution to the first component, than these two dimensions seem to have a high correlation. Later in this thesis, this technique will be used to analyze some measured data.

**Varimax Rotation with respect to PCA**

In order to further ease interpretation of the PCs, the rotated components are considered. This is a technique where the PCs are subject to an orthogonal rotation. As a result, the rotated components show a clearer pattern of loadings, where the variables either have a very low or high loading, thus showing either a negligible or a significant impact on the PC [38].

Most of the rationales for rotating factors are defended because this procedure simplifies the factor structure and therefore makes its interpretation easier and more reliable [1]. Certain criteria are made to identify a simple structure. According to these criteria, a matrix of loadings (where the rows correspond to the original variables and the columns to the factors) is simple if:

1. Each row contains at least one zero
2. For each column, there are at least as many zeros as there are columns (i.e., number of factors kept)
3. For any pair of factors, there are some variables with zero loadings on one factor and large loadings on the other factor
4. For any pair of factors, there is a sizable proportion of zero loadings
5. For any pair of factors, there is only a small number of large loadings

There exist different rotation techniques to transform principal components into simpler structures. The most frequently used is the **Varimax** rotation.

**Varimax** [1], which was developed by Kaiser, is indubitably the most popular rotation method by far. For varimax a simple solution means that each factor has a small number of large loadings and a large number of zero (or small) loadings. This simplifies the interpretation because, after a varimax rotation, each original variable tends to be associated with one (or a small number) of factors, and each factor represents only a small number of variables. In addition, the factors can often be interpreted from the opposition of few variables with positive loadings to few variables with negative loadings. Formally varimax searches for a rotation (i.e., a linear combination) of the original factors such that the variance of the loadings is maximized, which amounts to maximizing

\[ V = \sum (q_{j,l}^2 - \bar{q}_{j,l}^2)^2 \]

with \(q_{j,l}\) being the squared loading of the jth variable on the l factor, and \(\bar{q}_{j,l}\) being the mean of the squared loadings. For more information on Varimax see [1].

Varimax rotation is done upon selected PCS from PCA. Usually the PCS covering the most data variance are selected. They are distinguished by an eigenvalue higher than 1. However, sometimes a pc may have a low eigenvalue, but still covers much data variance (e.g. 7 percent). Therefore, in this thesis, those PCS will be selected which cover at least one percent of the data variance, until a total of at least 95 percent of the data is covered by the selected PCS.

**6.4 Regression Analysis: Logistic Regression**
Regression analysis is any statistical method where the mean of one or more random variables is predicted conditioned on other (measured) random variables. In particular, there is linear regression, logistic regression, Poisson regression, supervised learning, and unit-weighted regression. Regression analysis is more than curve fitting (choosing a curve that best fits given data points); it involves fitting a model with both deterministic and stochastic components. The deterministic component is called the predictor and the stochastic component is called the error term. Regression is usually posed as an optimization problem as we are attempting to find a solution where the error is at a minimum. The most common error measure that is used is the least squares: this corresponds to a Gaussian likelihood of generating observed data given the (hidden) random variable. Regression can be expressed as a maximum likelihood method of estimating the parameters of a model. However, for small amounts of data, this estimate can have high variance [14, 15 and 24].

The choice of a modeling technique for univariate analysis (and also the multivariate analysis that follows) is mostly driven by the nature of the dependent variable: its distribution, measurement scale, whether it is continuous or discrete.

Examples from the literature include:

- Logistic regression to predict the likelihood for an event to occur, e.g., fault detection.
- Ordinary least-squares regression often combined with monotonic transformation (logarithmic, quadratic) of the independent variables and/or dependent variable, to predict interval/ratio scale dependent variable.
- Negative binomial regression (of which Poisson regression is a special case) to predict discrete dependent variable that have low averages and whose distribution is skewed to the right.
- Parametric and non-parametric measures of correlation (Spearman Rho, Pearson r) are sometimes used.

Logistic regression is a standard classification technique based on maximum likelihood estimation. A multivariate logistic regression model is based on the following relationship:

\[
\pi(X_1, \cdots, X_n) = \frac{e^{(c_0 + c_1X_1 + \cdots + c_nX_n)}}{1 + e^{(c_0 + c_1X_1 + \cdots + c_nX_n)}}
\]
where \( \pi \) is the probability of the dependent variable \( Y \), the \( X_i \)'s are the independent variables/covariates. If the variable \( Y \) has only discrete values (for example, a Yes/No variable), logistic regression is preferred [15]. Therefore, later in this thesis logistic regression will be used for univariate and multivariate regression analysis.

The coefficients \( C_i \) are estimated through the maximization of a likelihood function \( L \), built in the usual fashion, i.e. as the product of the probabilities of the single observations, which are functions of the covariates (whose values are known in the observation) and the coefficients (which are the unknowns). For mathematical convenience,

\[
L - \ln[L], \quad \text{the log likelihood,}
\]

is usually the function to be maximized. This procedure assumes that all observations are statistically independent. We also want to assess the impact of the independent variable(s) on the dependent variable. In logistic regression, the regression coefficients, the \( C_i \)'s, cannot be easily interpreted for this purpose. Instead, a measure \( \Delta \psi \) is considered, which is based on the notion of the odds ratio.

The odds ratio

\[
\psi(X) = \frac{\pi(X)}{1 - \pi(X)}
\]

represents the ration between the probability an event being true and the probability of that event not being true. From this the following is derived:

\[
\Delta \psi = \frac{\psi(X + \sigma)}{\psi(X)}
\]

Here, \( \sigma \) stands for the standard deviation of measure \( X \). Therefore, \( \Delta \psi \) represents the reduction/increase in the odds ratio when the value \( X \) increases by one standard deviation. This is designed to provide an intuitive insight into the impact of independent variables. However,
some measures may contain very extreme outliers, which inflate the standard deviation of those measures. $\Delta \psi$ can then no longer be reasonably interpreted. Therefore, such outliers have to be excluded for the calculation of $\Delta \psi$.

The $\Delta \psi$ of each observation is calculated, and the mean of these values is used to evaluate the regression model [15, 24].

**Likelihood ratio Chi-square test**

To assess the statistical significance of each independent variable in the model, a likelihood ratio chi-square test is used. Let

$$1 = \ln[L]$$

be the log likelihood of the model derived by logical regression analysis, and $1_i$ be the log likelihood of the model without variable $X_i$. Assuming the null hypothesis that the true coefficient of $X_i$ is zero, the statistic

$$G = -2(l - l_i)$$

follows a chi-square distribution with one degree of freedom (denoted by $\chi^2(1)$) [15,35,37]. A test is done concerning

$$p = P(\chi^2(1) > G)$$

Where $p$ is the probability that the chi-square distribution of one degree of freedom of $l$ is larger than $G$. If $p$ is larger than some level of significance $\alpha$, (typically $\alpha=0.05$), the observed change in the log likelihood may well be due to chance, and $X_i$ is not considered significant. If $p<=\alpha$, the observed change in the log likelihood is unlikely to be due to chance, and $X_i$ is considered significant.

**Goodness of fit**
The global measure of goodness of fit we will use for such a model is assessed via $R^2$ – not to be confused with the least-square regression $R^2$ – they are built upon very different formulae, even though they both range between 0 and 1 and are similar from an intuitive perspective. The higher $R^2$, the higher the effect of the model’s explanatory variables, the more accurate the model. However, as opposed to the $R^2$ of least-square regression, high $R^2$s are rare for logistic regression. For this reason, the reader should not interpret logistic regression $R^2$s using the usual heuristics for least-square regression $R^2$s. Logistic regression $R^2$ is defined by the following ratio:

$$R^2 = \frac{LL_S - LL}{LL_S}$$, where

$LL$ is the log likelihood obtained by Maximum Likelihood Estimation of the model derived by logistic regression, and $LL_S$ is the log likelihood obtained by Maximum Likelihood Estimation of a model with the intercept $\beta_0$ only. By carrying out all the calculations, it can be shown that $LL_S$ is given by

$$LL_S = m_0 \ln\left(\frac{m_0}{m_0 + m_1}\right) + m_1 \ln\left(\frac{m_1}{m_0 + m_1}\right)$$

Where $m_0$ (resp., $m_1$) represent the number of observations for which the dependent variable is 0 (resp., high/1). Looking at the above formula, $LL_S/(m_0 + m_1)$ may be interpreted as the uncertainty associated with the distribution of the dependent variable $Y$, according to Information Theory concepts. It is the uncertainty left when the variable-less model is used. Likewise, $LL/(m_0 + m_1)$ may be interpreted as the uncertainty left when the model with the covariates is used. As a consequence, $(LL_S - LL)/(m_0 + m_1)$ may be interpreted as the part of uncertainty that is explained by the model. Therefore, the ratio

$$(LL_S - LL)/LL_S$$ may be interpreted as the proportion of uncertainty explained by the model [15].
6.5 Univariate Regression Analyses

When studying a dependent variable with the help of some independent variables, univariate regression analysis can be performed for each individual independent variable against the dependent variable, in order to determine if the measure is a potential useful predictor. Univariate regression analysis is conducted for two purposes [14]:

- To test the hypothesis that the independent variables have a significant statistical relationship with the dependent variable.
- To screen out measures which are not significant predictors in multivariate models (explained next). Only measures that are significant at significance level, say $\alpha = 0.05$, should be considered for the subsequent multivariate analysis.

6.6 The impact of design size: Spearman’s Rho coefficient

As mentioned before, the size of designs (e.g., class designs) is a necessary part of any predictive model. This is mostly justified by the fact that the size of any software artifact determines, to some extent, many of its external properties such as fault proneness or effort. But size only results in very poor prediction models, because any class bigger is size than another class will be predicted to contain more faults, in our case of building a fault prediction model.

For each measure we analyze its relationship to the size of the class, as measured based on design information. This is to determine empirically whether the measure, even though it is assumed to be any design structure measurement, is essentially measuring design size. This is important for several reasons. First, if a measure is strongly related to size, this might shed light on its relationship with fault-proneness: larger classes are more likely to contain faults. If a measure is strongly related to size, then it is likely that that measure does not contribute extra quality to the prediction model. Recall that we are also interested in increasing our understanding of OO code
and design quality, independently of its size. Size measures are selected showing the strongest relationship to fault proneness and most of its variance due to size: NMImp, the number of methods implemented in the class. Next the Spearman’s Rho coefficient between each design measure and size is calculated. A non-parametric measure of correlation is chosen, given the skewed distributions of the design measures that we usually observe [15]. This analysis is more refined but partially redundant with principal component analysis that will show us the measures being part of the same dimension as size measures.

**Spearman’s Rho coefficient**

In statistics, *Spearman's rho correlation coefficient*, named for Charles Spearman and often denoted by the Greek letter ρ (rho), is a non-parametric measure of correlation – that is, it assesses how well an arbitrary monotonic function could describe the relationship between two variables, without making any assumptions about the frequency distribution of the variables [2 and 31].

Correlation summarizes the strength of relationship between two variables, but it is important to remember that correlation is not causation. We require that the two variables, X and Y, are paired observations. Several different correlation coefficients can be calculated, but Spearman's Rho correlation coefficient is commonly used and advised [15, 24]. Spearman's coefficient requires data that are at least ordinal and the calculation is carried out on the ranks of the data. Each variable is ranked separately by putting the values of the variable in order and numbering them: the lowest value is given rank 1, the next lowest is given rank 2 and so on. If two data values for the variable are the same they are given averaged ranks, so if they would have been ranked 14 and 15 then they both receive rank 14.5. Spearman's rho correlation coefficient is used as a measure of linear relationship between two sets of ranked data, measures how tightly the ranked data clusters around a straight line.

In principle, ρ (rho) is simply a special case of the Pearson product-moment coefficient in which the data are converted to ranks before calculating the coefficient. In practice, however, a simpler procedure is normally used to calculate ρ. The raw scores are converted to ranks, and the differences $D$ between the ranks of each observation on the two variables are calculated. ρ is then given by [42]:


\[ \rho = 1 - \frac{6 \sum D^2}{N(N^2 - 1)} \]

where

\( D \) = the difference between the ranks of corresponding values of \( X \) and \( Y \), and  
\( N \) = the number of pairs of values.

Spearman's rho coefficient, like all other correlation coefficient, will take a value between 1 and +1. A positive correlation is one in which the ranks of both variables increase together. A negative correlation is one in which the ranks of one variable increase as the ranks of the other variable decrease. A correlation of +1 or 1 will arise if the relationship between the two variables is exactly linear. A correlation close to zero means there is no linear relationship between the ranks.

### 6.7 Deriving a prediction model: Multivariate Regression

#### 6.7.1 Prediction model construction

Multivariate logistic regression is performed to build prediction models of the fault-proneness of classes. For prediction model construction, a technique has to be used for selecting a subset of dependent variables for the prediction model. This analysis is conducted to determine how well we can predict the fault-proneness of classes, when the selected measures are used in combination. For the selection of measures to be used in the model, a strategy must be employed that

- Select an appropriate number of independent variables in the model. Over fitting a model increases the standard error of the model’s prediction, making the model more dependent on the data set it is based on and thus less generalizable [14,15].
- Reduces multicollinearity, i.e., independent variables which are highly correlated. High multicollinearity results in large standard errors for regression coefficient estimates and may affect the predictive power of the model. It also makes the estimate of the impact of one independent variable on the dependent variable difficult to derive from the model.

This study is exploratory in nature, that is, we do not have a strong theory that tells us which variables should be included in the prediction model and which not. There is no unique technique for selecting variables for model construction and many procedures have been suggested [23]. Not all of these procedures lead to the same solution when applied to the same problem, although for many problems they will achieve the same answer. In this situation, a stepwise selection process can be used, where prediction models are built in a stepwise manner, each time one variable enters or leaves the model. For other procedures see [23]. The two major stepwise selection processes used for regression model fitting are **forward selection** and **backward elimination** [15 and 23]. The general forward selection procedure starts with a model that includes the intercept only. Based on certain statistical criteria, variables are selected one at a time for inclusion in the model, until a stopping criterion is fulfilled. Similarly, the general backward elimination procedure starts with a model that includes all independent variables. Variables are selected one at a time to be deleted from the model, until a stopping criterion is fulfilled. Because of the large number of independent variables used in this study, the initial model in a backward selection process would contain too many variables and could not be interpreted in a meaningful way. Therefore, we opted for the forward selection procedure to build the prediction models. Also, forward selection is believed to be one of the best variable selection procedures and is recommended for usage [23]. It makes economical use of computer facilities, and it avoids working with more independent variables than are necessary while improving the equation at every stage.

**Forward selection**

This procedure starts off by choosing an equation containing only the intercept. In each step, all variables not already in the model are tested: the most significant variable is selected for inclusion in the model. If this causes a variable already in the model to become not significant, at a chosen significance level e.g. 0.10, it is deleted from the model. For the best variable to enter the model, it has to pass an entry test of significance, at a chosen significance level e.g. 0.05. The significance levels to enter and exit the model (0.05 and 0.10, respectively) are
stricter than those suggested normally, but recommended in [15]. We made this choice because it is an indirect means to control the number of variables in the final model. A predictor that may have been the best entry candidate at an earlier stage or in univariate analysis may, at a later stage, be superfluous because of the relationships between it and other variables now in the regression model. Eventually (unless the entry- and exit-levels are badly chosen to provide a cycling effect), when no variables in the current equation can be removed and the next best candidate cannot hold its place in the equation or the improvement of the equation is not significant (R2), the process stops.

Note that the forward selection does not necessarily find the best combination of variables (out of all possible combinations). However, it will result in a combination which comes close to the optimum solution. Figure 6.3 depicts a graphical overview of a prediction model construction. The independent and dependent variables are input to the multivariate model, which according to some stepwise selection process constructs a prediction model using on the other hand the estimated and true values of the dependent variables. According to the mapping of estimated values of the dependent variable and its true values, and depending on the significance of the independent variables at each step, the model is modified until no further improvement is possible by the used stepwise selection process.

![Figure 6.3: Graphical overview of prediction model construction](image)

6.7.2 Test of multicollinearity
Multivariate models should be tested for multicollinearity. The presence of multicollinearity makes the interpretation of the model difficult, as the impact of individual covariates on the dependent variable can no longer be judged independently from other covariates. In severe cases, multicollinearity results in inflated standard errors for the estimated coefficients, which renders predicted values of the model unreliable.

According to [32], tests for multicollinearity used in least-squares regression are also applicable in the context of logistic regression. They recommend a test based on the conditional number of the correlation matrix of the covariates in the model. This conditional number can conveniently be defined in terms of the eigenvalues of principal components as introduced in Section 3.2.2.

Let $X_1,\ldots, X_n$ be the covariates of our model. We perform a principal component analysis on these variables, and set $l_{\text{max}}$ to be the largest eigenvalue, $l_{\text{min}}$ the smallest eigenvalue of the principal components. The conditional number is then defined as

$$\lambda = \sqrt{\frac{l_{\text{max}}}{l_{\text{min}}}}$$

A large conditional number (i.e., discrepancy between minimum and maximum eigenvalue) indicates the presence of multicollinearity. A series of experiments showed that the degree of multicollinearity is harmful, and corrective actions should be taken, when the conditional number exceeds 30.

### 6.7.3 Goodness of fit

To evaluate the model’s goodness of fit, we apply the prediction model to the classes of our data set from which we derived the model. A class is classified fault-prone, if its predicted probability to contain a fault is higher than a certain threshold $p_0$. We select the threshold $p_0$ such that the percentage of classes being classified truly is maximized. That includes classes which are faulty and are also identified as faulty, and classes which are not faulty and not identified as faulty. In [15] a different function is maximized: that the percentage of classes being classified faulty is roughly the same as the percentage of classes that actually are fault-prone. The next step is to compare the predicted fault proneness of classes to their actual fault-proneness. We use the following measures of the goodness of fit of the prediction model:
• **Overall Completeness** - Assume we use the prediction model to select classes that are classified fault-prone for inspection. Further assume that inspections are 100% effective, i.e., all faults in a class are found during inspection. Overall Completeness is then defined as the number of classes classified fault-prone summed up with the number of classes classified non-faulty, divided by the total number of classes in the system. It is a measure of the percentage of classes identified right, faulty if they contain faults or non-faulty if they contain no faults, if we used the prediction model in the stated manner. Low Overall Completeness indicates that many classes are wrongly predicted to be faulty or to be not-faulty.

• **Completeness** - Again assume we use the prediction model to select classes that are classified fault-prone for inspection. Further assume that inspections are 100% effective, i.e., all faults in a class are found during inspection. Completeness is then defined as the number of faults in classes classified fault-prone, divided by the total number of faults in the system. It is a measure of the percentage of faults that would have been found if we used the prediction model in the stated manner. Low completeness indicates that many faults are not detected. These faults would then slip to subsequent development phases, where they are more expensive to correct. Counting the percentage of faults found is more precise than counting the percentage of fault-prone classes found. It ensures that detecting a class containing many faults contributes more to completeness than detecting a class with only one fault. Of course, faults can be more or less severe (e.g., measured by the effort required to fix a fault). It would be desirable to also account for the severity of faults when measuring completeness.

• **Correctness** - We can always increase the completeness of our prediction model by lowering the threshold $p_0$ used to classify classes as fault-prone ($p>p_0$). This causes more classes to be classified as fault-prone, thus completeness increases. However, the number of classes that are incorrectly being classified as fault-prone also increases. It is therefore important to consider the correctness of the prediction model. Correctness is the number of classes correctly classified as fault-prone, divided by the total number of classes classified as fault-prone. Low correctness means that a high percentage of the classes being classified as fault-prone do not actually contain a fault. We want correctness to be high, as inspections of classes that do not contain faults is a waste of resources.
A fourth measure of the goodness of fit is the $R^2$ statistic, as explained in Section 3.1. Unlike completeness, correctness, and Kappa, the definition of $R^2$ is specific to regression techniques based on maximum-likelihood estimation.

---

**Chapter 7**

**Data collection and analysis**

The analysis results, which are carried out in the way described in chapter 6, are given and interpreted in this chapter. First preliminary analysis is carried out, followed by principal component analysis, correlation to size and univariate regression analysis. The next step uses multivariate regression analysis to construct some fault-proneness prediction models, which are all evaluated. At each step, data of both, velocity and Tomcat, will be considered. In the end the hypotheses stated in chapter 5 will be investigated considering the outcome of the analysis results of this chapter.

**7.1 Preliminary results**
Table 7.1 presents the descriptive statistics of the Velocity data for the used measures. A total of 98 classes of Velocity are being evaluated. Columns “Max”, “75%”, “Med.”, “25%”, “Min.”, “Mean” and “Std Dev” state for each measure the maximum value, 75% quartile, median, 25% quartile, minimum, mean value, and standard deviation, respectively.

From this table, the following observations are made:

As in previous studies [3, 4 and 5], the mean (and total) values for dynamic import coupling metrics (e.g., IC_OC) are equal to their corresponding dynamic export coupling measure (e.g., EC_OC). This confirms the symmetry property discussed earlier, stating that the total number of messages (or method invocations) sent is always equal to the total number of messages (or method invocations) received. This property also holds for the xx_xCC measures (IC_OCC, IC_CCC, EC_OCC and EC_CCC), which should not be a surprise.
<table>
<thead>
<tr>
<th>Metric</th>
<th>Mean</th>
<th>Median</th>
<th>Min</th>
<th>Max</th>
<th>StdDev</th>
</tr>
</thead>
<tbody>
<tr>
<td>EC_CC</td>
<td>858</td>
<td>112</td>
<td>9</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>EC_CCC</td>
<td>514</td>
<td>52</td>
<td>6</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>NG_EC_CCC</td>
<td>3140</td>
<td>66</td>
<td>55</td>
<td>32.5</td>
<td>10</td>
</tr>
<tr>
<td>CBO</td>
<td>649</td>
<td>55</td>
<td>7</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>NG_CBO</td>
<td>4152</td>
<td>94</td>
<td>61</td>
<td>52.5</td>
<td>17</td>
</tr>
<tr>
<td>CBOImp</td>
<td>377</td>
<td>47</td>
<td>5</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>NG_CBOImp</td>
<td>399</td>
<td>54</td>
<td>4</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>CBOExp</td>
<td>275</td>
<td>50</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>NG_CBOExp</td>
<td>516</td>
<td>50</td>
<td>3</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>NMImp</td>
<td>1028</td>
<td>167</td>
<td>9</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>NAImp</td>
<td>304</td>
<td>33</td>
<td>4</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 7.1: Descriptive Statistics of the Velocity data for the used metrics

Also measures of strength “distinct Methods” seem to be higher than measures of strength “distinct classes”, which themselves are higher than measures of strength “distinct class-to-class”. This, because for a collection of distinct methods (m1, c1, m2, c2), where their m1, c1 and c2 part all the same, but their m2 part differ, are mapped to just one distinct class (m1, c1, c2). And for a collection of distinct classes (m1, c1, c2), of which their c1 and c2 part are all equal, but their m1 part differs, are mapped to just one distinct Class-to-Class measure (c1, c2).

Most of the measures, excluding the dynamic neighbor measures and NG_CBO, show a large standard deviation and mean values that are higher than the median values, which imply that the distribution is skewed with a tail towards larger values. The descriptive statistics for the size measures do not show any interesting or surprising trends. Also, for most of the measures, there are large differences between the lower 25th percentile, the median and the 75th percentile, thus showing strong variations. This seems to be less for all static coupling measures (including static neighbor measures). The standard deviations for all measures are all high enough to include them for further analysis.

It must be noted, that the Total value of CBO is less than the sum of CBOImp and CBOExp. This because CBOImp and CBOExp have some elements in common.

Table 7.2 presents the descriptive statistics of the Tomcat data for the used measures, in the same ways as in table 7.1. A total of 147 classes of Tomcat are being evaluated.
The same conclusions derived from table 7.1 hold here. This data confirms the previous observation that measurements of strength “distinct Methods” are higher than measures of strength “distinct classes”, which themselves are higher than measures of strength “distinct class-to-class”.

Here for all measures, the mean value is higher than their median values. This was earlier not the case with some neighbor coupling measures. For most dynamic coupling measures, their mean values regarding Tomcat data are lower than those of Velocity. This indicates that for the mean class in Tomcat, less distinct messages/method invocations are send/received, which may result in different results regarding regression analysis.

The median values of the dynamic neighbor coupling measures in table 7.1 show large values, whereas those in table 7.2 are low in comparison. This means that the distribution of these dynamic neighbor coupling measures is more skewed towards the higher values in the tomcat data. This may have an effect on regression analysis on the tomcat data being different.

Further, the variance seems good enough for all measures: there are large differences between their 25th and 75th percentiles.

<table>
<thead>
<tr>
<th>Descriptive Statistics for metric :</th>
<th>Total</th>
<th>Max</th>
<th>75%</th>
<th>Med</th>
<th>25%</th>
<th>Min</th>
<th>Mean</th>
<th>Std Deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>IC_OM</td>
<td>2476</td>
<td>229</td>
<td>14</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>13</td>
<td>33</td>
</tr>
<tr>
<td>IC_OC</td>
<td>1336</td>
<td>147</td>
<td>9</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>6.99</td>
<td>15.4</td>
</tr>
<tr>
<td>IC_OCC</td>
<td>725</td>
<td>53</td>
<td>6</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>3.8</td>
<td>7.46</td>
</tr>
<tr>
<td>NG_IC_OCC</td>
<td>3522</td>
<td>108</td>
<td>46</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>18.4</td>
<td>27.7</td>
</tr>
<tr>
<td>IC_CM</td>
<td>2422</td>
<td>217</td>
<td>14</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>12.7</td>
<td>31.9</td>
</tr>
<tr>
<td>IC_CC</td>
<td>1383</td>
<td>150</td>
<td>10</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>7.24</td>
<td>15.9</td>
</tr>
<tr>
<td>IC_CCC</td>
<td>765</td>
<td>57</td>
<td>7</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>4.01</td>
<td>7.7</td>
</tr>
<tr>
<td>NG_IC_CCC</td>
<td>3844</td>
<td>115</td>
<td>43</td>
<td>7</td>
<td>0</td>
<td>0</td>
<td>20.1</td>
<td>28.9</td>
</tr>
<tr>
<td>EC_OM</td>
<td>2476</td>
<td>320</td>
<td>17</td>
<td>5</td>
<td>2</td>
<td>0</td>
<td>13</td>
<td>29.5</td>
</tr>
<tr>
<td>EC_OC</td>
<td>1336</td>
<td>115</td>
<td>8.7</td>
<td>4</td>
<td>1</td>
<td>0</td>
<td>6.99</td>
<td>13.9</td>
</tr>
<tr>
<td>EC_OCC</td>
<td>725</td>
<td>37</td>
<td>6</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>3.8</td>
<td>5.38</td>
</tr>
</tbody>
</table>
Table 7.2: Descriptive Statistics of the Tomcat data for the used metrics

<table>
<thead>
<tr>
<th>Metric</th>
<th>NG_EC_OCC</th>
<th>EC_CM</th>
<th>EC_CC</th>
<th>EC_CCC</th>
<th>NG_EC_CCC</th>
<th>CBO</th>
<th>NG_CBO</th>
<th>CBOImp</th>
<th>NG_CBOImp</th>
<th>CBOExp</th>
<th>NG_CBOExp</th>
<th>NMImp</th>
<th>NAImp</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>3581</td>
<td>66</td>
<td>41</td>
<td>18</td>
<td>4</td>
<td>0</td>
<td>18.7</td>
<td>19.7</td>
<td>12.7</td>
<td>28.5</td>
<td>7.24</td>
<td>15.9</td>
<td>4.01</td>
</tr>
<tr>
<td></td>
<td>2422</td>
<td>295</td>
<td>16</td>
<td>7</td>
<td>2</td>
<td>0</td>
<td>12.7</td>
<td>28.5</td>
<td>4.01</td>
<td>5.66</td>
<td>4.01</td>
<td>5.66</td>
<td>4.01</td>
</tr>
<tr>
<td></td>
<td>1383</td>
<td>171</td>
<td>8.7</td>
<td>4</td>
<td>2</td>
<td>0</td>
<td>7.24</td>
<td>15.9</td>
<td>4.01</td>
<td>5.66</td>
<td>4.01</td>
<td>5.66</td>
<td>4.01</td>
</tr>
<tr>
<td></td>
<td>765</td>
<td>49</td>
<td>6</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>4.01</td>
<td>5.66</td>
<td>4.01</td>
<td>5.66</td>
<td>4.01</td>
<td>5.66</td>
<td>4.01</td>
</tr>
<tr>
<td></td>
<td>3904</td>
<td>72</td>
<td>41</td>
<td>19</td>
<td>7</td>
<td>0</td>
<td>20.4</td>
<td>21</td>
<td>4.01</td>
<td>5.66</td>
<td>4.01</td>
<td>5.66</td>
<td>4.01</td>
</tr>
<tr>
<td></td>
<td>1209</td>
<td>86</td>
<td>11</td>
<td>5</td>
<td>3</td>
<td>1</td>
<td>8.2</td>
<td>9.6</td>
<td>4.01</td>
<td>5.66</td>
<td>4.01</td>
<td>5.66</td>
<td>4.01</td>
</tr>
<tr>
<td></td>
<td>9124</td>
<td>155</td>
<td>100</td>
<td>61</td>
<td>15.3</td>
<td>0</td>
<td>62.1</td>
<td>44.9</td>
<td>4.01</td>
<td>5.66</td>
<td>4.01</td>
<td>5.66</td>
<td>4.01</td>
</tr>
<tr>
<td></td>
<td>731</td>
<td>35</td>
<td>8</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>5</td>
<td>6</td>
<td>4.01</td>
<td>5.66</td>
<td>4.01</td>
<td>5.66</td>
<td>4.01</td>
</tr>
<tr>
<td></td>
<td>983</td>
<td>68</td>
<td>8.7</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>6.7</td>
<td>11.7</td>
<td>4.01</td>
<td>5.66</td>
<td>4.01</td>
<td>5.66</td>
<td>4.01</td>
</tr>
<tr>
<td></td>
<td>523</td>
<td>86</td>
<td>3.7</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>3.6</td>
<td>8</td>
<td>4.01</td>
<td>5.66</td>
<td>4.01</td>
<td>5.66</td>
<td>4.01</td>
</tr>
<tr>
<td></td>
<td>927</td>
<td>49</td>
<td>10</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>6.3</td>
<td>8.2</td>
<td>4.01</td>
<td>5.66</td>
<td>4.01</td>
<td>5.66</td>
<td>4.01</td>
</tr>
<tr>
<td></td>
<td>2716</td>
<td>168</td>
<td>23</td>
<td>11</td>
<td>6</td>
<td>1</td>
<td>18.5</td>
<td>20.7</td>
<td>4.01</td>
<td>5.66</td>
<td>4.01</td>
<td>5.66</td>
<td>4.01</td>
</tr>
<tr>
<td></td>
<td>1000</td>
<td>48</td>
<td>8</td>
<td>4</td>
<td>2</td>
<td>0</td>
<td>6.8</td>
<td>7.8</td>
<td>4.01</td>
<td>5.66</td>
<td>4.01</td>
<td>5.66</td>
<td>4.01</td>
</tr>
</tbody>
</table>

7.2 Principal Component Analysis

In [3, 4 and 5] Principal Component Analysis (PCA) results have shown that coupling for the dynamic measures IC_OD, IC_OM, IC_OC, IC_CD, IC_CM, IC_CC, EC_OD, EC_OM, EC_OC, EC_CD, EC_CM and EC_CC is divided along four dimensions: IC_Ox, IC_Cx, EC_Ox and EC_Cx, belonging to identical components and therefore capturing similar properties.

To test whether the same results are found for the dynamic measures of [3, 4 and 5], PCA was done on the Velocity data containing the following variables: IC_OM, IC_OC, IC_CM, IC_CC, EC_OM, EC_OC, EC_CM and EC_CC. The results after Varimax rotation on the first 4 components are shown in table 7.3. As can be seen, the PCA results confirm that coupling is divided among 4 dimensions.
Table 7.3: PCA results for selected measures.

The dynamic measures IC_OCC, IC_CCC, EC_OCC and EC_CCC are derived from their corresponding dynamic measures with same scope, granularity and entity of measurement. Therefore they are assumed to also participate in their components.

To test this statement, PCA was extended with these 4 metrics, and the results can be seen in table 7.4, which satisfies the hypothesis made.
Table 7.4: PCA results for selected measures.

PCA was done including all the metrics studied in this project. These are 16 dynamic coupling metrics, 4 static coupling metrics and 2 size metrics. The results are put in table 7.5.
<table>
<thead>
<tr>
<th>Measure</th>
<th>NG_EC_CCC</th>
<th>-0.01</th>
<th>-0.10</th>
<th>-0.04</th>
<th>0.06</th>
<th>0.00</th>
<th>-0.93</th>
<th>-0.04</th>
<th>0.06</th>
</tr>
</thead>
<tbody>
<tr>
<td>CBO</td>
<td>0.39</td>
<td>-0.16</td>
<td>0.02</td>
<td>0.11</td>
<td>-0.27</td>
<td>0.11</td>
<td>-0.07</td>
<td>0.16</td>
<td>-0.11</td>
</tr>
<tr>
<td>NG_CBO</td>
<td>-0.22</td>
<td>0.10</td>
<td>0.42</td>
<td>0.01</td>
<td>-0.49</td>
<td>0.07</td>
<td>-0.06</td>
<td>-0.33</td>
<td>0.01</td>
</tr>
<tr>
<td>CBOImp</td>
<td>0.47</td>
<td>0.16</td>
<td>0.05</td>
<td>-0.09</td>
<td>-0.01</td>
<td>0.01</td>
<td>-0.03</td>
<td>0.07</td>
<td>-0.08</td>
</tr>
<tr>
<td>NG_CBOImp</td>
<td>0.09</td>
<td>-0.06</td>
<td>-0.07</td>
<td>0.03</td>
<td>0.02</td>
<td>0.02</td>
<td>-0.01</td>
<td>-0.88</td>
<td>-0.03</td>
</tr>
<tr>
<td>CBOExp</td>
<td>0.10</td>
<td>-0.35</td>
<td>-0.02</td>
<td>0.22</td>
<td>-0.35</td>
<td>0.13</td>
<td>-0.07</td>
<td>0.14</td>
<td>-0.07</td>
</tr>
<tr>
<td>NG_CBOExp</td>
<td>0.02</td>
<td>0.07</td>
<td>-0.19</td>
<td>-0.10</td>
<td>-0.72</td>
<td>-0.05</td>
<td>0.09</td>
<td>0.03</td>
<td>0.07</td>
</tr>
<tr>
<td>NMImp</td>
<td>0.18</td>
<td>0.06</td>
<td>-0.09</td>
<td>-0.06</td>
<td>-0.04</td>
<td>-0.52</td>
<td>0.04</td>
<td>-0.03</td>
<td>-0.05</td>
</tr>
<tr>
<td>NAImp</td>
<td>-0.05</td>
<td>-0.11</td>
<td>0.06</td>
<td>0.08</td>
<td>0.03</td>
<td>-0.80</td>
<td>-0.05</td>
<td>0.04</td>
<td>0.01</td>
</tr>
</tbody>
</table>

Table 7.5: PCA results on Velocity data for all measures.

Based on the analysis of the loadings associated with each measure within each of the rotated components, the PCs are interpreted as follows:

**PC1**: IC_CM, IC_CC, IC_CCC, CBO and CBOImp have the highest loading in this pc. The first three measures represent dynamic import coupling with the class level as level of measurement. The other two measures are static coupling measures, of which the last consists of import coupling only, along with the dynamic measures, and CBO contains both, import and export coupling. The dynamic measures are constructed from the collection of method invocations because of their level of measurement. This maps messages to the real invocation taking place due to inheritance, polymorphism and dynamic binding. The two static measures, instead of being as precise as the dynamic measures, use the base of the class which implements the invoked methods. But both kind of measures their elements contain classes which implement the invoked method. This, together with all measures containing import coupling will be the interpretation of this pc.

**PC2**: EC_CM, EC_CC and EC_CCC have the highest loadings in this component, and represent dynamic export coupling with the class level as entity of measurement.
**PC3:** NG_IC_OCC and NG_IC_CCC are the important metrics in this pc. These are dynamic neighbor coupling metrics and represent for each class the import coupling between his direct environment and the rest of the system, where his direct environment stands for the collection of classes the subject class is coupled (IC_OCC and IC_CCC respectively) to. NG_EC_OCC also has a small loading in this pc. This metric represents for each class the export coupling between his direct environment and the rest of the system, where his direct environment stands for the collection of classes the subject class is coupled (EC_OCC) to. What these metrics have in common is the coupling between a class his direct environment and the rest of the classes. The other export neighbor coupling metric NG_EC_CCC is not significant present in this pc, and has its own pc: 7, where NG_EC_OCC is also part of, but very small. This observation regarding NG_EC_OCC is strange.

**PC4:** EC_OM, EC_OC and EC_OCC have high loadings in this pc. They represent dynamic export coupling with the object level as entity of measurement.

**PC5:** NG_CBO and NG_CBOExp represent his pc. This pc is interpreted as static export coupling between the direct environment of a class and the rest of the classes in the system. By direct environment of a class is meant the collection of classes to which the subject class is coupled directly thru CBO and CBOExp respectively. Here, CBOExp only considers static export coupling and CBO considers both import and export coupling. It is the export coupling CBO and CBOExp have both in common.

**PC6:** NAImp has the lead in this component. NMImp also has some loading in this PC. NAImp and NAImp are both size measures, which will be the interpretation of this pc. As it suspects, both size measures capture almost the same data. Later, this will be investigated more.

**PC7:** NG_EC_CCC is the important metric in this pc. This is a dynamic export neighbor coupling metrics and represent for each class the export coupling between his direct environment and the rest of the system, where his direct environment stands for the collection of classes the subject class is coupled (thru EC_CCC) to. NG_EC_OCC, which is based on EC_OCC, also has a small loading in this pc.
**PC8**: NG_CBOImp has the only high value in this pc. This measurement measures the static import neighbor coupling between the direct environment of a class and the rest of the classes in the system. By direct environment of a class is meant the collection of classes to which the subject class is coupled directly thru CBOImp.

**PC9**: IC_OM, IC_OC and IC_OCC have the highest loading in this pc. They represent dynamic import coupling with the object level as entity of measurement, which is derived from the collection of messages.

<table>
<thead>
<tr>
<th></th>
<th>PC1</th>
<th>PC2</th>
<th>PC3</th>
<th>PC4</th>
<th>PC5</th>
<th>PC6</th>
<th>PC7</th>
<th>PC8</th>
<th>PC9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Eigenvalue</td>
<td>10.95</td>
<td>4.30</td>
<td>1.88</td>
<td>1.58</td>
<td>1.35</td>
<td>1.16</td>
<td>0.88</td>
<td>0.52</td>
<td>0.28</td>
</tr>
<tr>
<td>Percent</td>
<td>0.46</td>
<td>0.18</td>
<td>0.08</td>
<td>0.07</td>
<td>0.06</td>
<td>0.04</td>
<td>0.04</td>
<td>0.02</td>
<td>0.01</td>
</tr>
<tr>
<td>Cum Percent</td>
<td>0.46</td>
<td>0.64</td>
<td>0.72</td>
<td>0.78</td>
<td>0.84</td>
<td>0.88</td>
<td>0.92</td>
<td>0.94</td>
<td>0.95</td>
</tr>
<tr>
<td>IC_OM</td>
<td><strong>0.46</strong></td>
<td>-0.10</td>
<td>0.03</td>
<td>0.03</td>
<td>-0.01</td>
<td>-0.01</td>
<td>-0.04</td>
<td>-0.03</td>
<td>0.03</td>
</tr>
<tr>
<td>IC_OC</td>
<td><strong>0.28</strong></td>
<td>-0.06</td>
<td>-0.09</td>
<td>0.03</td>
<td>-0.28</td>
<td>0.08</td>
<td>0.06</td>
<td>0.02</td>
<td>0.04</td>
</tr>
<tr>
<td>IC_OCC</td>
<td><strong>0.39</strong></td>
<td>-0.09</td>
<td>-0.01</td>
<td>-0.06</td>
<td>-0.08</td>
<td>0.01</td>
<td>-0.03</td>
<td>0.12</td>
<td>0.02</td>
</tr>
<tr>
<td>NG_IC_OCC</td>
<td>0.05</td>
<td>-0.06</td>
<td>-0.03</td>
<td>0</td>
<td>-0.13</td>
<td>0.05</td>
<td>-0.09</td>
<td><strong>0.60</strong></td>
<td>0.01</td>
</tr>
<tr>
<td>IC_CM</td>
<td><strong>0.47</strong></td>
<td>0.04</td>
<td>0.04</td>
<td>0.03</td>
<td>0.12</td>
<td>-0.04</td>
<td>0</td>
<td>-0.06</td>
<td>-0.01</td>
</tr>
<tr>
<td>IC_CC</td>
<td><strong>0.31</strong></td>
<td>0.14</td>
<td>-0.05</td>
<td>0.05</td>
<td>-0.04</td>
<td>0.03</td>
<td>0.11</td>
<td>-0.03</td>
<td>-0.02</td>
</tr>
<tr>
<td>IC_CCC</td>
<td><strong>0.41</strong></td>
<td>0.08</td>
<td>0.02</td>
<td>-0.07</td>
<td>0.13</td>
<td>-0.04</td>
<td>0.02</td>
<td>0.12</td>
<td>-0.03</td>
</tr>
<tr>
<td>NG_IC_CCC</td>
<td>0.06</td>
<td>0.09</td>
<td>0.02</td>
<td>0.01</td>
<td>0.07</td>
<td>-0.01</td>
<td>0</td>
<td><strong>0.58</strong></td>
<td>-0.07</td>
</tr>
<tr>
<td>EC_OM</td>
<td>0</td>
<td>0.02</td>
<td>-0.05</td>
<td>0.03</td>
<td><strong>-0.54</strong></td>
<td>0.10</td>
<td>0.11</td>
<td>-0.09</td>
<td>0.05</td>
</tr>
<tr>
<td>EC_OC</td>
<td>-0.07</td>
<td>0.05</td>
<td>-0.02</td>
<td>-0.01</td>
<td><strong>-0.54</strong></td>
<td>-0.05</td>
<td>-0.02</td>
<td>0.07</td>
<td>0</td>
</tr>
<tr>
<td>EC_OCC</td>
<td>-0.04</td>
<td>0.01</td>
<td>0.17</td>
<td>-0.06</td>
<td><strong>-0.48</strong></td>
<td>-0.13</td>
<td>-0.05</td>
<td>0.05</td>
<td>-0.12</td>
</tr>
<tr>
<td>NG_EC_OCC</td>
<td>0.03</td>
<td>-0.12</td>
<td><strong>0.60</strong></td>
<td>0.01</td>
<td>-0.13</td>
<td>0.08</td>
<td>-0.09</td>
<td>0.01</td>
<td>0.16</td>
</tr>
<tr>
<td>EC_CM</td>
<td>0.03</td>
<td><strong>0.55</strong></td>
<td>-0.06</td>
<td>0.02</td>
<td>-0.03</td>
<td>0.12</td>
<td>0.04</td>
<td>-0.05</td>
<td>0.08</td>
</tr>
<tr>
<td>EC_CC</td>
<td>-0.03</td>
<td><strong>0.60</strong></td>
<td>-0.05</td>
<td>0.02</td>
<td>0.01</td>
<td>0</td>
<td>-0.05</td>
<td>0.06</td>
<td>0.04</td>
</tr>
<tr>
<td>Measure</td>
<td>Value1</td>
<td>Value2</td>
<td>Value3</td>
<td>Value4</td>
<td>Value5</td>
<td>Value6</td>
<td>Value7</td>
<td>Value8</td>
<td></td>
</tr>
<tr>
<td>-----------</td>
<td>--------</td>
<td>--------</td>
<td>--------</td>
<td>--------</td>
<td>--------</td>
<td>--------</td>
<td>--------</td>
<td>--------</td>
<td></td>
</tr>
<tr>
<td>EC_CCC</td>
<td>0.49</td>
<td>0.11</td>
<td>-0.05</td>
<td>-0.04</td>
<td>-0.10</td>
<td>-0.02</td>
<td>0.01</td>
<td>-0.08</td>
<td></td>
</tr>
<tr>
<td>NG_EC_CCC</td>
<td>0.74</td>
<td>-0.02</td>
<td>0.11</td>
<td>-0.03</td>
<td>0.09</td>
<td>-0.01</td>
<td>-0.12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CBO</td>
<td>0.01</td>
<td>-0.01</td>
<td>-0.02</td>
<td>0.18</td>
<td>0.01</td>
<td>-0.65</td>
<td>0.07</td>
<td>0.01</td>
<td></td>
</tr>
<tr>
<td>NG_CBO</td>
<td>-0.18</td>
<td>-0.05</td>
<td>0.10</td>
<td>0.20</td>
<td>0.08</td>
<td>-0.03</td>
<td>0.28</td>
<td>0.34</td>
<td></td>
</tr>
<tr>
<td>CBOImp</td>
<td>-0.03</td>
<td>-0.02</td>
<td>-0.03</td>
<td>0.47</td>
<td>0.06</td>
<td>-0.10</td>
<td>0.23</td>
<td>0.13</td>
<td></td>
</tr>
<tr>
<td>NG_CBOImp</td>
<td>0.05</td>
<td>0</td>
<td>0.01</td>
<td>0.80</td>
<td>-0.06</td>
<td>0.06</td>
<td>-0.25</td>
<td>-0.12</td>
<td></td>
</tr>
<tr>
<td>CBOExp</td>
<td>0.02</td>
<td>-0.01</td>
<td>-0.02</td>
<td>-0.13</td>
<td>-0.05</td>
<td>-0.70</td>
<td>-0.08</td>
<td>-0.05</td>
<td></td>
</tr>
<tr>
<td>NG_CBOExp</td>
<td>0.08</td>
<td>0.06</td>
<td>0</td>
<td>-0.11</td>
<td>0</td>
<td>-0.03</td>
<td>-0.09</td>
<td>-0.14</td>
<td></td>
</tr>
<tr>
<td>NMImp</td>
<td>0.10</td>
<td>0.01</td>
<td>0.07</td>
<td>0.10</td>
<td>-0.05</td>
<td>0.02</td>
<td>0.53</td>
<td>-0.26</td>
<td></td>
</tr>
<tr>
<td>NAImp</td>
<td>-0.01</td>
<td>-0.03</td>
<td>-0.06</td>
<td>-0.10</td>
<td>-0.04</td>
<td>0.01</td>
<td>0.66</td>
<td>0.06</td>
<td></td>
</tr>
</tbody>
</table>

Table 7.6: PCA results on the Tomcat data for all measures.

Table 7.6 shows the PCA results on the Tomcat data. Globally the measures are stacked together in a pc as in table 7.5. Both Size measures are as previously representing the same pc. All IC_Ox measures are together in one pc, as is the case for all IC_Cx, EC_Ox, EC_Cx and size measure. The differences are that the IC_Ox and IC_Cx measures share the same pc, which is interpreted as dynamic import coupling (see section 7.7 for a better explanation). Also, both CBO and CBOExp clearly are in one pc, which was not the case earlier. An explanation for this is that CBO captures both, import and export coupling. In previous analysis (Velocity) it’s shared the same pc with CBOImp, now it switched to CBOExp.

Dynamic neighbor coupling was divided in two pcs in table 7.1. One pc for import and the other for export neighbor coupling. However, NG_EC_OCC was more part of the former one than the later. In this table, they are clearly divided in a static import and export neighbor coupling pc.

### 7.3 Correlation to Size
In this section we analyze the correlation of the design measures to the size of the class [45]. We first measure the size of the class design as the number of methods (non-inherited and overridden) that are implemented in the class, and as a second size measure the size of the class design as the number of attributes (non-inherited) that a class contains. The correlation of each design measure with size is expressed in terms of its Spearman Rho coefficient and the corresponding p-value, as is explained in chapter 6. Below in table 7.7 are the results presented for the velocity data. Measures with a low p-value (lower than 0.01) are considered significant to that particular size measure and there value is made bold in the table.

IC_CM, IC_CC, EC_OM, EC_CM, EC_CC, EC_CCC, NG_EC_CCC, CBO and CBOImp have p-values lower than 0.01 when tested for their relation with NMImp, and are thus considered to be significantly related to NMImp. However, the Rho coefficients for most of these measures are below 0.5, i.e., their relationship to size (NMImp) is not that strong. Still their relationship with size may have some influence in regression analysis. EC_CM, EC_CC and EC_CCC however have high Rho values, and are considered to be strongly related to NMImp.

<table>
<thead>
<tr>
<th>Measure</th>
<th>NMImp Rho</th>
<th>NMImp p-value</th>
<th>NAImp Rho</th>
<th>NAImp p-value</th>
</tr>
</thead>
<tbody>
<tr>
<td>IC_OM</td>
<td>0.2539</td>
<td>0.0116</td>
<td>0.1419</td>
<td>0.1635</td>
</tr>
<tr>
<td>IC_OC</td>
<td>0.2423</td>
<td>0.0162</td>
<td>0.1078</td>
<td>0.2909</td>
</tr>
<tr>
<td>IC_OCC</td>
<td>0.1589</td>
<td>0.1181</td>
<td>0.0831</td>
<td>0.4158</td>
</tr>
<tr>
<td>NG_IC_OCC</td>
<td>0.0294</td>
<td>0.7741</td>
<td>-0.1489</td>
<td>0.1433</td>
</tr>
<tr>
<td>IC_CM</td>
<td>0.3927</td>
<td><strong>0.0001</strong></td>
<td>0.2461</td>
<td>0.0146</td>
</tr>
<tr>
<td>IC_CC</td>
<td>0.3717</td>
<td><strong>0.0002</strong></td>
<td>0.2185</td>
<td>0.0307</td>
</tr>
<tr>
<td>IC_CCC</td>
<td>0.2281</td>
<td>0.0239</td>
<td>0.1275</td>
<td>0.2108</td>
</tr>
<tr>
<td>NG_IC_CCC</td>
<td>0.1891</td>
<td>0.0622</td>
<td>0.0509</td>
<td>0.6184</td>
</tr>
<tr>
<td>EC_OM</td>
<td>0.3319</td>
<td><strong>0.0008</strong></td>
<td>0.1502</td>
<td>0.1399</td>
</tr>
<tr>
<td>EC_OC</td>
<td>0.2422</td>
<td>0.0163</td>
<td>0.0616</td>
<td>0.5466</td>
</tr>
<tr>
<td>EC_OCC</td>
<td>0.1286</td>
<td>0.2069</td>
<td>-0.0651</td>
<td>0.5245</td>
</tr>
<tr>
<td>NG_EC_OCC</td>
<td>0.0912</td>
<td>0.3716</td>
<td>-0.0178</td>
<td>0.8616</td>
</tr>
<tr>
<td>EC_CM</td>
<td><strong>0.7964</strong></td>
<td>&lt; <strong>0.0001</strong></td>
<td><strong>0.6872</strong></td>
<td>&lt; <strong>0.0001</strong></td>
</tr>
</tbody>
</table>
### Table 7.7: Spearman’s rho coefficient and p-value on the Velocity data for each measure

<table>
<thead>
<tr>
<th>Measure</th>
<th>NMImp Rho</th>
<th>NMImp p-value</th>
<th>NAImp Rho</th>
<th>NAImp p-value</th>
</tr>
</thead>
<tbody>
<tr>
<td>IC_OM</td>
<td></td>
<td></td>
<td>IC_OC</td>
<td></td>
</tr>
<tr>
<td>IC_OC</td>
<td></td>
<td></td>
<td>IC_OCC</td>
<td></td>
</tr>
<tr>
<td>IC_CM</td>
<td>0.4436</td>
<td>&lt;0.0001</td>
<td>0.4873</td>
<td>&lt;0.0001</td>
</tr>
<tr>
<td>IC_CC</td>
<td>0.4660</td>
<td>&lt;0.0001</td>
<td>0.5013</td>
<td>&lt;0.0001</td>
</tr>
</tbody>
</table>

EC_CM, EC_CC, EC_CCC, CBO and CBOExp show very low p-values when tested for the significance with respect to their relation with NAImp. CBO, CBOExp and EC_CCC however have a low Rho value. EC_CM and EC_CC are considered to be strongly related to NAImp. They also showed a strong relation towards NMImp.

It is also noticeable that only the EC_Cx measures show a strong relation towards size, not the EC_Ox measures. This may be because the Size measures do not account for inherited methods and variables, and on the other hand dynamic measures with the object level of measurement do include elements derive from messages of which the object’s class to which a message is send (or from which a message is send) may have inherited the method to which the message is send (or from which a message is send).
Table 7.8: Spearman’s rho coefficient and p value on the Tomcat data for each measure

<table>
<thead>
<tr>
<th>Measure</th>
<th>Rho</th>
<th>p value</th>
<th>Rho</th>
<th>p value</th>
<th>Rho</th>
<th>p value</th>
</tr>
</thead>
<tbody>
<tr>
<td>IC_CCC</td>
<td>0.4049</td>
<td>&lt;0.0001</td>
<td>0.4854</td>
<td>&lt;0.0001</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NG_IC_CCC</td>
<td>0.2968</td>
<td>0.0003</td>
<td>0.4152</td>
<td>&lt;0.0001</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EC_OM</td>
<td>0.2824</td>
<td>0.0005</td>
<td>0.2523</td>
<td>0.0020</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EC_OC</td>
<td>0.2374</td>
<td>0.0038</td>
<td>0.2068</td>
<td>0.0120</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EC_OCC</td>
<td>0.1910</td>
<td>0.0205</td>
<td>0.1443</td>
<td>0.0813</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NG_EC_OCC</td>
<td>0.1116</td>
<td>0.1785</td>
<td>0.1229</td>
<td>0.1168</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EC_CM</td>
<td>0.5214</td>
<td>&lt;0.0001</td>
<td>0.4790</td>
<td>&lt;0.0001</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EC_CC</td>
<td>0.5106</td>
<td>&lt;0.0001</td>
<td>0.4683</td>
<td>&lt;0.0001</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EC_CCC</td>
<td>0.4601</td>
<td>&lt;0.0001</td>
<td>0.4057</td>
<td>&lt;0.0001</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NG_EC_CCC</td>
<td>0.3308</td>
<td>&lt;0.0001</td>
<td>0.3405</td>
<td>&lt;0.0001</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CBO</td>
<td>0.5241</td>
<td>&lt;0.0001</td>
<td>0.5584</td>
<td>&lt;0.0001</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NG_CBO</td>
<td>0.5321</td>
<td>&lt;0.0001</td>
<td>0.5587</td>
<td>&lt;0.0001</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CBOImp</td>
<td>0.5138</td>
<td>&lt;0.0001</td>
<td>0.5233</td>
<td>&lt;0.0001</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NG_CBOImp</td>
<td>0.2480</td>
<td>0.0025</td>
<td>0.2602</td>
<td>0.0015</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CBOExp</td>
<td>0.3106</td>
<td>0.0001</td>
<td>0.3101</td>
<td>0.0001</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NG_CBOExp</td>
<td>0.1816</td>
<td>0.0277</td>
<td>0.1050</td>
<td>0.2055</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 7.8 presents the Spearman’s Rho coefficients and p-values for the Tomcat data.

All dynamic import coupling measures their rho values for NMImp stayed around the same value, or increased. All of them seem to be significant related to NAImp, whereas according to the Velocity data none were. Also, excluding IC_OCC, they all show now also a significant relation to NMImp. There Rho values are however not high enough to mark them as strong related to size.

The dynamic export coupling measures show also a significant relation to size, but their rho values have decreased. Especially EC_CM and EC_CC who according to the Velocity data seemed to be strong related to Size.

CBO, NG_CBO and CBOImp are not only significant related to both size measures, but also, in comparison to the Velocity data results, has their rho value increased till above 0.5. However there rho value is still not high enough to consider them as strong related to size, if the size measures show to be good predictors for fault-proneness, then the static coupling measures their relation to fault-proneness according to the Tomcat data might be stronger compared to their
relation with fault-proneness according to the Velocity data. In the next section this will be investigated.

NMImp and NAImp are counting very different attributes of a class. The number of methods (non-inherited and overridden) implemented in a class versus the number of attributes (non-inherited) declared at class-level (accessible to all elements in that class) in a class. Still EC_CM, EC_CC and EC_CCC showed a strong relation towards both size measures. From PCA analysis it seem that this is not strange since these 2 size measures were part of the same pc, and so having almost the same underlying dimensions in their data variance. I analyzed their relation with the Spearman Rho’s coefficient in order to see if their data variance has something in common. Table 7.9 contains the results of both Velocity and Tomcat.

<table>
<thead>
<tr>
<th>Measure</th>
<th>Rho</th>
<th>p-value</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAImp</td>
<td>0.6895</td>
<td>&lt; 0.0001</td>
<td>Velocity 1.2</td>
</tr>
<tr>
<td>NAImp</td>
<td>0.7597</td>
<td>&lt; 0.0001</td>
<td>Tomcat 4.1.0</td>
</tr>
</tbody>
</table>

Table 7.9: relation between NMImp and NAImp

NMImp and NAImp have are very significant related to each other according to both data available, Tomcat and Velocity, and the high Rho values also shows that they are strongly related to each other. This is a confirmation of what PCA showed in previous sections. Their relation is interpreted as follows: The more attributes a class contains, the more functionality the class represents. Chances are high that more methods are needed to household this increased functionality. Also, it becomes a common programming attitude that, when using the java language (which is used by Velocity and Tomcat), for private attributes of a class, if they need to be accessed or changed from outside the class, a get and a put method are usually made available.
In multivariate regression analysis this relation between NMImp and NAImp may have some influence: because the data they measure has almost the same underlying variance/structure, the chance is high that if size is significant related to fault-proneness, only one of these two measures will be included in a multivariate model.

### 7.4 Univariate Regression Analyses

The results of the univariate analysis on the Velocity data are summarized in Table 7.10. For each measure, the regression coefficient and standard error is provided (Columns "Coeff." and "Std Err"), the $R^2$ and $\Delta \psi$ value (as defined in chapter 6), and the statistical significance (p-value), which is the probability that the coefficient is different from zero by chance. Because all measures had enough variance (see descriptive statistics earlier), they are all included for univariate regression analysis.

<table>
<thead>
<tr>
<th>Measure</th>
<th>R²</th>
<th>Coeff.</th>
<th>Std Err</th>
<th>$\Delta \psi$</th>
<th>p-value</th>
</tr>
</thead>
<tbody>
<tr>
<td>IC_OM</td>
<td>0.0485</td>
<td>0.0236</td>
<td>0.0124</td>
<td>2.3032</td>
<td>0.0108</td>
</tr>
<tr>
<td>IC_OC</td>
<td>0.0496</td>
<td>0.0398</td>
<td>0.0194</td>
<td>2.3555</td>
<td>0.0099</td>
</tr>
<tr>
<td>IC_OCC</td>
<td>0.0257</td>
<td>0.0528</td>
<td>0.0301</td>
<td>1.4873</td>
<td>0.0640</td>
</tr>
<tr>
<td>NG_IC_OCC</td>
<td>0.0039</td>
<td>0.0047</td>
<td>0.0065</td>
<td>1.1601</td>
<td>0.4670</td>
</tr>
<tr>
<td>IC_CM</td>
<td>0.2096</td>
<td>0.1386</td>
<td>0.0418</td>
<td>Inf</td>
<td>&lt;0.0001</td>
</tr>
<tr>
<td>IC_CC</td>
<td>0.2066</td>
<td>0.1908</td>
<td>0.0557</td>
<td>Inf</td>
<td>&lt;0.0001</td>
</tr>
<tr>
<td>IC_CCC</td>
<td>0.1290</td>
<td>0.1939</td>
<td>0.0623</td>
<td>3.4630</td>
<td>&lt;0.0001</td>
</tr>
<tr>
<td>NG_IC_CCC</td>
<td>0.0548</td>
<td>0.0218</td>
<td>0.0083</td>
<td>1.7799</td>
<td>0.0065</td>
</tr>
<tr>
<td>EC_OM</td>
<td>0.0263</td>
<td>0.0216</td>
<td>0.0123</td>
<td>1.4996</td>
<td>0.0610</td>
</tr>
<tr>
<td>EC_OC</td>
<td>0.0126</td>
<td>0.0206</td>
<td>0.0165</td>
<td>1.3164</td>
<td>0.1939</td>
</tr>
<tr>
<td>EC_OCC</td>
<td>0.0067</td>
<td>0.0293</td>
<td>0.0317</td>
<td>1.2170</td>
<td>0.3451</td>
</tr>
<tr>
<td>NG_EC_OCC</td>
<td>0.0085</td>
<td>0.0101</td>
<td>0.0095</td>
<td>1.2473</td>
<td>0.2829</td>
</tr>
<tr>
<td>EC_CM</td>
<td>0.0622</td>
<td>0.0449</td>
<td>0.0215</td>
<td>3.0070</td>
<td>0.0039</td>
</tr>
<tr>
<td>Measure</td>
<td>EC_CC</td>
<td>EC_CCC</td>
<td>NG_EC_CCC</td>
<td>CBO</td>
<td>NG_CBO</td>
</tr>
<tr>
<td>------------------</td>
<td>-------</td>
<td>--------</td>
<td>------------</td>
<td>-------</td>
<td>--------</td>
</tr>
<tr>
<td></td>
<td>0.0456</td>
<td>0.0246</td>
<td>0.0143</td>
<td>0.0995</td>
<td>0.0048</td>
</tr>
<tr>
<td></td>
<td>0.0524</td>
<td>0.0642</td>
<td>0.0121</td>
<td>0.2069</td>
<td>0.0066</td>
</tr>
<tr>
<td></td>
<td>0.0278</td>
<td>0.0417</td>
<td>0.0088</td>
<td>0.0705</td>
<td>0.0083</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 7.10: univariate regression results on Velocity data for all measures

Not all measures have a significant relationship with fault-proneness (at max p-value 0.05). The non-significant measures their p-value is marked with the red color. They are excluded for further analysis in the rest of this thesis concerning the analysis of the same data (Velocity in this case). Of all the export coupling measures (static and dynamic), only EC_CM and EC_CC are found to be significant. And even these two measures have very low R2 values and coefficients. Of the significant measures, not all are strongly related to fault-proneness, which can be seen by a small R2 and coefficient value. The standard error of all measures seems low enough.

Out of all the dynamic coupling measures, only IC_CM and IC_CC have a high R2 and high coefficients. IC_CCC also has a high coefficient and \( \Delta \psi \), but a lower R2. CBOImp is the only static coupling measure which has a strong relation with fault-proneness. It’s R2 value is less compared to IC_CM and IC_CC, but its coefficient is almost twice as high. On the other hand, the coefficient value is hard to interpret. NAImp seems to have a stronger relation with fault-proneness than its size partner NMImp. Apparently, despite their high correlation, they still differ in some way. NG_CBOImp is the only neighbor coupling measure which shows some mild relation with fault-proneness, when considering its low R2, but its relative high coefficient. Overall, NAImp seems the best univariate measure for fault-proneness.
The univariate regression was executed on data filtered from significant univariate outliers. Table 7.11 contains the number of extreme outliers detected and the number of significant considered outliers for each measure. There are a total of 98 data points, where each data point has values for the 24 metrics considered in this thesis, and the actual faults detected thru time in the class which the data point represents. At most, for NG_CBOExp, 15 extreme outliers were detected, and for all measures at most only one significant outlier is detected. The significance level was set at 5% change in the R2 value of the univariate regression model. This means that an outlier was considered as significant if when excluding that outlier from univariate analysis, the change in the R2 value of the univariate model was higher or equal to 5%. Only significant outliers were removed for further analysis.

<table>
<thead>
<tr>
<th>Measure</th>
<th>Outliers</th>
<th>Significant</th>
</tr>
</thead>
<tbody>
<tr>
<td>IC_OM</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>IC_OC</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>IC_OCC</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>NG_IC_OCC</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>IC_CM</td>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>IC_CC</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>IC_CCC</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>NG_IC_CCC</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>EC_OM</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>EC_OC</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>EC_OCC</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>NG_EC_OCC</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Table 7.11: univariate outliers in Velocity data for all measures

<table>
<thead>
<tr>
<th>Measure</th>
<th>EC_CM</th>
<th>EC_CC</th>
<th>EC_CCC</th>
<th>NG_EC CCC</th>
<th>CBO</th>
<th>NG_CBO</th>
<th>CBOImp</th>
<th>NG_CBOImp</th>
<th>CBOExp</th>
<th>NG_CBOExp</th>
<th>NMImp</th>
<th>NAImp</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>6</td>
<td>6</td>
<td>5</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>1</td>
<td>5</td>
<td>12</td>
<td>15</td>
<td>7</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 7.12 presents univariate regression results for Tomcat data, and table 7.13 contains the number of extreme outliers and the number of outliers classified as significant.

The results here differ very much with that derived from the Velocity data. Analyzing the significance of the measures, all EC_Ox measures and NG_EC_OCC are still not significant to fault-proneness. For dynamic export coupling measures, only EC_CM and EC_CCC are further investigated, along with the export neighbor coupling measure NG_EC_CCC.

Where previously 2 measures of the dynamic import coupling measures and dynamic import neighbor coupling were found not significant, here all dynamic import measures have low p-values. The R2 value of IC_CC is again the largest among the dynamic import coupling measures. The dynamic import neighbor coupling measures also have high R2 values, which was not the case earlier. The R2 value of the IC_Cx measures are higher than the r2 value of their object level of measurement variant, the IC_Ox measures.

The size measures have not changed much. NMImp’s R2 has gone up, but still NAImp is stronger related to fault-proneness.
The static coupling measures have changed much. CBO’s R2 value is just below the R2 value of IC_CC, but its coefficient is higher. CBOImp has the highest R2 value and coefficient of all measures, even compared to the size measures. Earlier was observed that for the Tomcat data, these measures have a much stronger relation with size when compared with the Velocity data. This may be an explanation of the change in how strong they are related to fault-proneness according to the Tomcat data.

NG_CBO has become strongly related to fault-proneness, which was also the case for the dynamic import neighbor coupling measures.

<table>
<thead>
<tr>
<th>Measure</th>
<th>R2</th>
<th>Coeff.</th>
<th>Std Err</th>
<th>Δψ</th>
<th>p-value</th>
</tr>
</thead>
<tbody>
<tr>
<td>IC_OM</td>
<td>0.1327</td>
<td>0.0560</td>
<td>0.0163</td>
<td>7.9141</td>
<td>&lt;0.0001</td>
</tr>
<tr>
<td>IC_OC</td>
<td>0.1401</td>
<td>0.0996</td>
<td>0.0259</td>
<td>5.5471</td>
<td>&lt;0.0001</td>
</tr>
<tr>
<td>IC_OCC</td>
<td>0.1237</td>
<td>0.1795</td>
<td>0.0487</td>
<td>4.3901</td>
<td>&lt;0.0001</td>
</tr>
<tr>
<td>NG_IC_OCC</td>
<td>0.1546</td>
<td>0.0368</td>
<td>0.0077</td>
<td>2.9964</td>
<td>&lt;0.0001</td>
</tr>
<tr>
<td>IC_CM</td>
<td>0.1726</td>
<td>0.0809</td>
<td>0.0211</td>
<td>17.922</td>
<td>&lt; 0.0001</td>
</tr>
<tr>
<td>IC_CC</td>
<td>0.1975</td>
<td>0.1436</td>
<td>0.0331</td>
<td>12.753</td>
<td>&lt;0.0001</td>
</tr>
<tr>
<td>IC_CCC</td>
<td>0.1670</td>
<td>0.2367</td>
<td>0.0564</td>
<td>7.4661</td>
<td>&lt;0.0001</td>
</tr>
<tr>
<td>NG_IC_CCC</td>
<td>0.2178</td>
<td>0.0458</td>
<td>0.0086</td>
<td>4.1305</td>
<td>&lt;0.0001</td>
</tr>
<tr>
<td>EC_OM</td>
<td>0.0156</td>
<td>0.0117</td>
<td>0.0079</td>
<td>1.4668</td>
<td>0.0757</td>
</tr>
<tr>
<td>EC_OC</td>
<td>0.0104</td>
<td>0.0170</td>
<td>0.0127</td>
<td>1.2974</td>
<td>0.1478</td>
</tr>
<tr>
<td>EC_OCC</td>
<td>0.0046</td>
<td>0.0294</td>
<td>0.0310</td>
<td>1.1770</td>
<td>0.3348</td>
</tr>
<tr>
<td>NG_EC_OCC</td>
<td>0.0152</td>
<td>0.0147</td>
<td>0.0084</td>
<td>1.3421</td>
<td>0.0781</td>
</tr>
</tbody>
</table>
Table 7.12: univariate regression results on Tomcat data for all measures

<table>
<thead>
<tr>
<th>Measure</th>
<th>Outliers</th>
<th>Significant</th>
</tr>
</thead>
<tbody>
<tr>
<td>IC_OM</td>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td>IC_OC</td>
<td>9</td>
<td>0</td>
</tr>
<tr>
<td>IC_OCC</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>NG_IC_OCC</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>IC_CM</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>IC_CC</td>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>IC_CCC</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>NG_IC_CCC</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>EC_OM</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>EC_OC</td>
<td>8</td>
<td>1</td>
</tr>
</tbody>
</table>

The number of detected extreme outliers is not much different from the ones found in the Velocity data. But it is nice to notice that out of the dynamic measures, only from dynamic export measures, one outlier for each was marked as significant.
### 7.5 Prediction model construction: Multivariate Regression Analysis

In this section a number of multivariate prediction models are investigated, built from different subsets of the measures analyzed so far. The main goal is to investigate how significant different subsets of these (neighbor) coupling metrics are in predicting fault-proneness of classes, when combined with other static metrics like size (NIAmp and NMImp).

The strategy used in this chapter is to build some fault-proneness models containing measures chosen thru multivariate regression analysis, using both, Velocity and Tomcat data. First the Velocity data will be used, thereafter Tomcat data will be analyzed. For each constructed model, its goodness of fit will be examined using the data from which the model is derived (Velocity data or Tomcat data), and in addition its goodness of fit will be examined using data which was not used for constructing the model (Tomcat data and Velocity data respectively). This will show some insight in how good the model predicts faults in software environments other than the one

<table>
<thead>
<tr>
<th>Metric</th>
<th>Count</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>EC_OCC</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>NG_EC_OCC</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>EC_CM</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>EC_CC</td>
<td>11</td>
<td>1</td>
</tr>
<tr>
<td>EC_CCC</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>NG_EC_CCC</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>CBO</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>NG_CBO</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>CBOImp</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>NG_CBOImp</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>CBOExp</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>NG_CBOExp</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>NMImp</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>NAImp</td>
<td>5</td>
<td>0</td>
</tr>
</tbody>
</table>

*Table 7.13: univariate outliers in Tomcat data for all measures*
used to construct the model. For each model, its determined beta values and threshold value will be used for analyzing both data.

The first model built will be based on size measures only, and will be used as a base model to be compared with. Afterwards, the rest of the measures will be allowed to enter the model and I will compare its goodness of fit with the size-only model. Because of the usage of the forward selection procedure, a way has to be chosen to determine at different stages of the algorithm whether the being constructed model is improving or not. The R2 aspect of a model was chosen to be maximized. But this however does not have to result in a model which has the best completeness and/or correctness, because it does not try out all possible combination of the available variables. Also, forward selection selects at each step the next most significant variable not yet selected. It is possible that other variables which are less significant, but still good enough (p-value < 0.05), together add more predictive power/quality to the model than the most significant variable at the moment, but might not be selected in a later stage because of their changed p-value (p-value > 0.05) when entering the model in that stage. Because of these drawbacks of the forward selection procedure, a collection of models will be built, allowing all the measures in the beginning, and removing at subsequent steps one or more variables found most significant in previous steps.

Earlier, in chapter 6, a procedure was explained to detect multivariate outliers, and test their significance. In multivariate outlier detection, data of each observation (in this thesis “each class”) plays a role in determining if one observation/class is an outlier. This data is multidimensional since it consists of different measures for that class. In forward selection, sub-partitions of these measures are used at each step to finally result in an almost best model for prediction. At these steps, models with a subgroup of measures are taken to build a temporary prediction model with these selected measures. If multivariate outliers were computed before this procedure, this would not be consistent, because variables not in the current evaluated model played a role in detecting those outliers. That is why in building multivariate models, at each step, the multivariate outliers for that model are computed, taking in consideration only those dimensions of the data which measurements they represent and if those measurements are part of the current models.

When evaluating the models, the reader should ask him/her self the following questions:
- Are dynamic coupling design measures fault-proneness predictors that are complementary to design size measures?
- How much more accurate is a model including the more difficult to collect dynamic coupling measures? If it is not significantly better, then the additional effort of calculating these more expensive measures instead of some easily collected static coupling measures and size measures would not be justified.
- Are import coupling measures (static and dynamic) more useful predictors of external quality attributes of software systems/components than their export coupling variants?
- Do dynamic coupling measures with different entity of measurement, object versus class level, make a difference in predicting external quality attributes of software systems/components?
- Do dynamic coupling measures with different strength, dynamic messages (excluded in this project) versus distinct methods versus distinct classes, make a difference in predicting external quality attributes of software systems/components?
- Are neighbor coupling measures (dynamic and static variants) useful predictors of fault-proneness?

In section 7.7 the answers to these questions will be given. First, the constructed models will be presented.

**Size Model construction using Velocity data**

The size measures in this thesis only consist of NMImp and NAImp, where the later has shown a strong relation with fault-proneness in univariate regression analysis. When including these two measures for building a multivariate model of size measures only, only NAImp is selected by the procedure. When forcing both measures in the model, the p-value of NMImp was 0.1, and that was the reason forward selection did not include this measure. The Size model thus consists only of NAImp, and its attributes can be seen in table 7.14.

<table>
<thead>
<tr>
<th>Measure</th>
<th>Coefficient</th>
<th>Std. Err.</th>
<th>p-value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intercept</td>
<td>-1.2639</td>
<td>0.3180</td>
<td>0</td>
</tr>
<tr>
<td>NAImp</td>
<td>0.5109</td>
<td>0.1333</td>
<td>&lt; 0.0001</td>
</tr>
</tbody>
</table>

Table 7.14: The Size model
The log-likelihood of the model is -50.0463, the conditional number is just 1, which is understandable because only one variable is in the model, and the R2 is 0.2597.

We applied this model to the 98 classes of Velocity in order to compare the predicted and actual fault-proneness of the classes. A class was classified ‘predicted fault-prone’, if its predicted probability to contain a fault, computed by the Size model, is higher than 0.5625. There were 5 univariate outliers detected, of which none was significant, and not excluded. The table below contains the goodness of fit of the Size model for different data. The beta values of table 7.14 are used together with the determined value, 0.5625, to assign a class to be faulty or not.

<table>
<thead>
<tr>
<th>Overall Completeness</th>
<th>Completeness</th>
<th>Correctness</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>76%</td>
<td>67%</td>
<td>82%</td>
<td>Velocity</td>
</tr>
<tr>
<td>67%</td>
<td>90%</td>
<td>63%</td>
<td>Tomcat</td>
</tr>
</tbody>
</table>

*Table 7.15: Goodness of fit of the Size model*

As explained earlier, **Overall Completeness** is the number of classes correctly classified fault-prone summed up with the number of classes correctly classified non-faulty, divided by the total number of classes in the system. **Completeness** is the number of faults in classes correctly classified fault-prone, divided by the total number of faults in the system, and **Correctness** is the number of classes correctly classified as fault-prone, divided by the total number of classes classified as fault-prone.

Although this model only contains one size measure, its goodness of fit is still quite high. When using the model to analyze the Tomcat data, the overall completeness drops almost 10 %, the correctness almost 20% but the completeness rises with 23%. So for the Tomcat data more faults are seemed to be covered (having in mind that multiple of these faults can be of the same class), but more non faulty classes are classified as faulty.

**Model 1: built from all measures using Velocity data**
Next all measures all included, except those who had a high p-value at univariate regression analysis (and are marked red in table 7.8). The current variable pool consists of the following measures:

IC_OM, IC_OC, IC_CM, IC_CC, IC_CCC, NG_IC_CCC, EC_CM, EC_CC, CBO, CBOImp, NG_CBOImp, NMImp and NAImp.

The constructed model, from now on entitled as model 1, with its participating measures is listed in table 7.16.

<table>
<thead>
<tr>
<th>Measure</th>
<th>Coefficient</th>
<th>Std. Err.</th>
<th>p-value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intercept</td>
<td>-2.9589</td>
<td>0.6473</td>
<td>0</td>
</tr>
<tr>
<td>CBOImp</td>
<td>0.4975</td>
<td>0.1461</td>
<td>&lt;0.0001</td>
</tr>
<tr>
<td>NAImp</td>
<td>0.6269</td>
<td>0.1675</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 7.16: Model 1 and its participating measures

Model 1 includes, besides the size measure NAImp, the static coupling measure CBOImp. The log-likelihood of the model is -41.6134, the conditional number is just 1.7079, which does not indicate any problem, and the R2 is 0.3844, which is much higher than the Size model. We also applied this model to the 98 classes of Velocity in order to compare the predicted and actual fault-proneness of the classes. A class was classified ‘predicted fault-prone’, if its predicted probability to contain a fault, computed by this model, is higher than 0.4750. There were 6 multivariate outliers detected, of which none was significant, so not excluded. The table below contains the goodness of fit of this model. Like the previous model, when analyzing Tomcat data, both, overall completeness and correctness drops, but completeness rises, as was the case with the size-only model.

<table>
<thead>
<tr>
<th>Overall Completeness</th>
<th>Completeness</th>
<th>Correctness</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>84%</td>
<td>77%</td>
<td>91%</td>
<td>Velocity</td>
</tr>
<tr>
<td>80%</td>
<td>99%</td>
<td>74%</td>
<td>Tomcat</td>
</tr>
</tbody>
</table>

Table 7.17: Goodness of fit of Model 1
Alternative Models constructed using Velocity data

Next, some alternative models will be constructed with limited variables as input. The next variable pool which will be available for model construction contains all measures as earlier excluding CBOImp, which was the most significant variable selected in the previous model construction besides NAImp:

IC_OM, IC_OC, IC_CM, IC_CC, IC_CCC, NG_IC_CCC, EC_CM, EC_CC, CBO, NG_CBOImp, NMImp and NAImp.

The following tables list the information regarding Model 2:

<table>
<thead>
<tr>
<th>Measure</th>
<th>Coefficient</th>
<th>Std. Err.</th>
<th>p-value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intercept</td>
<td>-2.0876</td>
<td>0.4815</td>
<td>0</td>
</tr>
<tr>
<td>IC_OM</td>
<td>-0.1203</td>
<td>0.0682</td>
<td>0.0014</td>
</tr>
<tr>
<td>IC_CC</td>
<td>0.4107</td>
<td>0.1651</td>
<td>0</td>
</tr>
<tr>
<td>NAImp</td>
<td>0.4604</td>
<td>0.1435</td>
<td>0</td>
</tr>
</tbody>
</table>

Table7.18: Model 2 and its participating measures

Model 2 includes, besides the size measure NAImp, also IC_OM and IC_CC, of which the later is more significant (which confirms univariate regression analysis). The log-likelihood of the model is -38.3693, the conditional number is 6.4741, which does not indicate any problem, and the R2 is 0.4258. For applying this model to the 98 classes of Velocity, a class was classified ‘predicted fault-prone’, if its predicted probability to contain a fault, computed by this model, is higher than 0.4500. There were 12 multivariate outliers detected, of which 1 were found significant and therefore excluded. The table below contains the goodness of fit of this model. Like the previous models, when analyzing Tomcat data, both, overall completeness and correctness drops, but completeness rises.

<table>
<thead>
<tr>
<th>Overall Completeness</th>
<th>Completeness</th>
<th>Correctness</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>86%</td>
<td>77%</td>
<td>94%</td>
<td>Velocity</td>
</tr>
</tbody>
</table>
The next model is constructed from a variable pool containing all measures as in the earlier step, excluding IC_OM:

IC_OC, IC_CM, IC_CC, IC_CCC, NG_IC_CCC, EC_CM, EC_CC, CBO, NG_CBOImp, NMImp and NAImp. The resulting model, model 3, its aspects are covered by the tables below:

The following tables list the information regarding Model 3:

<table>
<thead>
<tr>
<th>Measure</th>
<th>Coefficient</th>
<th>Std. Err.</th>
<th>p-value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intercept</td>
<td>-2.4223</td>
<td>0.5736</td>
<td>0</td>
</tr>
<tr>
<td>IC_OC</td>
<td>-0.2699</td>
<td>0.1359</td>
<td>0.0129</td>
</tr>
<tr>
<td>IC_CC</td>
<td>0.6771</td>
<td>0.2313</td>
<td>0</td>
</tr>
<tr>
<td>NAImp</td>
<td>0.4481</td>
<td>0.1831</td>
<td>0.0010</td>
</tr>
</tbody>
</table>

Model 3 includes, besides the size measure NAImp, also IC_OC and IC_CC, of which the later is more significant (and which also confirms univariate regression analysis). The log-likelihood of the model is \(-36.4398\), the conditional number is \(5.0944\), which does not indicate any problem, and the R\(^2\) is \(0.4236\), which is just a bit lower than model 2. For applying this model to the 98 classes of Velocity, a class was classified ‘predicted fault-prone’, if its predicted probability to contain a fault, computed by this model, is higher than \(0.4625\). There were 13 multivariate outliers detected, of which only 6 were found significant and therefore excluded. The table below contains the goodness of fit of this model. Again, like the previous model, when analyzing Tomcat data, both, overall completeness and correctness drops, but completeness rises.

<table>
<thead>
<tr>
<th>Overall Completeness</th>
<th>Completeness</th>
<th>Correctness</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>87%</td>
<td>77%</td>
<td>97%</td>
<td>Velocity</td>
</tr>
<tr>
<td>77%</td>
<td>95%</td>
<td>72%</td>
<td>Tomcat</td>
</tr>
</tbody>
</table>

Table 7.19: Goodness of fit of Model 2

Table 7.20: Model 3 and its participating measures

Table 21: Goodness of fit of Model 3
The next model is constructed from a variable pool containing all measures as in the earlier step, excluding IC_CC:

IC_OC, IC_CM, IC_CCC, NG_IC_CCC, EC_CM, EC_CC, CBO, NG_CBOImp, NMImp and NAImp. The resulting model, model 4, its aspects are covered by the tables below:

<table>
<thead>
<tr>
<th>Measure</th>
<th>Coefficient</th>
<th>Std. Err.</th>
<th>p-value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intercept</td>
<td>-1.6201</td>
<td>0.3836</td>
<td>0</td>
</tr>
<tr>
<td>IC_CM</td>
<td>0.0853</td>
<td>0.0402</td>
<td>0.0014</td>
</tr>
<tr>
<td>NAImp</td>
<td>0.3886</td>
<td>0.1234</td>
<td>0</td>
</tr>
</tbody>
</table>

*Table 7.22: Model 4 and its participating measures*

Model 4 includes, besides the size measure NAImp, the static coupling measure IC_CM. The log-likelihood of the model is -44.9116, the conditional number is just 7.6226, which does not indicate any problem, and the R2 is 0.3356. For applying this model to the 98 classes of Velocity, a class was classified ‘predicted fault-prone’, if its predicted probability to contain a fault, computed by this model, is higher than 0.3375. There were 11 multivariate outliers detected, of which none were found significant. The table below contains the goodness of fit of this model. Like the previous models, when analyzing Tomcat data, both, overall completeness and correctness drops, but completeness rises.

<table>
<thead>
<tr>
<th>Overall Completeness</th>
<th>Completeness</th>
<th>Correctness</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>81%</td>
<td>81%</td>
<td>81%</td>
<td>Velocity</td>
</tr>
<tr>
<td>67%</td>
<td>98%</td>
<td>61%</td>
<td>Tomcat</td>
</tr>
</tbody>
</table>

*Table 7.23: Goodness of fit of Model 4*

The last model is constructed from the variable pool containing all measures as in the earlier step, excluding all dynamic import coupling measures:

NG_IC_CCC, EC_CM, EC_CC, CBO, NG_CBOImp, NMImp and NAImp. The following tables list the information regarding Model 5:
<table>
<thead>
<tr>
<th>Measure</th>
<th>Coefficient</th>
<th>Std. Err.</th>
<th>p-value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intercept</td>
<td>-1.8122</td>
<td>0.4314</td>
<td>0</td>
</tr>
<tr>
<td>NG_CBOImp</td>
<td>0.2082</td>
<td>0.1095</td>
<td>0.0077</td>
</tr>
<tr>
<td>NAImp</td>
<td>0.5394</td>
<td>0.1439</td>
<td>0</td>
</tr>
</tbody>
</table>

**Table 7.24: Model 5 and its participating measures**

Model 5 includes, besides the size measure NAImp, the static neighbor coupling measure NG_CBO. The log-likelihood of the model is **-46.4984**, the conditional number is just **1.7807**, which does not indicate any problem, and the R2 is **0.3122**. For applying this model to the 98 classes of Velocity, a class was classified ‘predicted fault-prone’, if its predicted probability to contain a fault, computed by this model, is higher than **0.3625**. There were 15 multivariate outliers detected, of which none was found significant. The table below contains the goodness of fit of this model. Again, like in the previous models, when analyzing Tomcat data, both, overall completeness and correctness drops, but completeness rises.

<table>
<thead>
<tr>
<th>Overall Completeness</th>
<th>Completeness</th>
<th>Correctness</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>80%</td>
<td>81%</td>
<td>78%</td>
<td>Velocity</td>
</tr>
<tr>
<td>65%</td>
<td>97%</td>
<td>59%</td>
<td>Tomcat</td>
</tr>
</tbody>
</table>

**Table 7.25: Goodness of fit of Model 5**

As seen, all the previous models have been built using the Velocity data. When analyzing Tomcat data with the derived model, overall completeness and correctness would drop compared to when analyzing Velocity data with the same model, and completeness would rise. This was the case for all models built using Velocity data. As it seems, when analyzing the Tomcat data with prediction models derived from velocity data, more faults are covered (where multiple faults may be of the same class), but more non faulty classes are classified as faulty (lower correctness and overall completeness).

Next, prediction models will be constructed in the same way as earlier, but using the **Tomcat** data.
**Size Model construction using Tomcat data**

Like with the Velocity data, the model consists only of NAImp, and its attributes can be seen in table 7.26.

<table>
<thead>
<tr>
<th>Measure</th>
<th>Coefficient</th>
<th>Std. Err.</th>
<th>p-value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intercept</td>
<td>-1.6722</td>
<td>0.3340</td>
<td>0</td>
</tr>
<tr>
<td>NAImp</td>
<td>0.3058</td>
<td>0.0614</td>
<td>&lt; 0.0001</td>
</tr>
</tbody>
</table>

*Table 7.26: The Size model*

The log-likelihood of the model is **-74.2254**, the conditional number is just **1**, which is understandable because only one variable is in the model, and the R$^2$ is **0.2713**.

We applied this model to the 147 classes of Tomcat in order to compare the predicted and actual fault-proneness of the classes. A class was classified ‘predicted fault-prone’, if its predicted probability to contain a fault, computed by the Size model, is higher than **0.6125**. There were 8 univariate outliers detected, of which none was significant, so not excluded. The table below contains the goodness of fit of the Size model. Both completeness evaluators are lower when analyzing Velocity data with the derived model. Correctness however increases.

<table>
<thead>
<tr>
<th>Overall Completeness</th>
<th>Completeness</th>
<th>Correctness</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>76%</td>
<td>74%</td>
<td>85%</td>
<td>Tomcat</td>
</tr>
<tr>
<td>70%</td>
<td>51%</td>
<td>100%</td>
<td>Velocity</td>
</tr>
</tbody>
</table>

*Table 7.27: Goodness of fit of the Size model*

Although this model only contains one size measure, its goodness of fit is still quite high. It is somewhat better than the Size model constructed from the Velocity data.

**Model 1: built from all measures using Tomcat data**
Next all measures all included, except those who had a high p-value at univariate regression analysis (and are marked red in table 7.12). The current variable pool consists of the following measures:

IC_OM, IC_OC, IC_OCC, NG_IC_OCC, IC_CM, IC_CC, IC_CCC, NG_IC_CCC, EC_CM, EC_CCC, NG_EC_CCC, CBO, NG_CBO, CBOImp, NG_CBOImp, NMImp and NAImp.

The constructed model, from now on entitled as model 1, with its participating measures is listed in table 7.28.

<table>
<thead>
<tr>
<th>Measure</th>
<th>Coefficient</th>
<th>Std. Err.</th>
<th>p-value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intercept</td>
<td>-2.4680</td>
<td>0.4181</td>
<td>0</td>
</tr>
<tr>
<td>CBOImp</td>
<td>0.3786</td>
<td>0.0837</td>
<td>0</td>
</tr>
<tr>
<td>NAImp</td>
<td>0.1938</td>
<td>0.0695</td>
<td>0.0011</td>
</tr>
</tbody>
</table>

Table 7.28: Model 1 and its participating measures

Model 1 includes, besides the size measure NAImp, the static coupling measure CBOImp. The log-likelihood of the model is -56.1141, the conditional number is just 2.1176, which does not indicate any problem, and the R2 is 0.4491, which is much higher than the Size model. We also applied this model to the 147 classes of Tomcat in order to compare the predicted and actual fault-proneness of the classes. A class was classified ‘predicted fault-prone’, if its predicted probability to contain a fault, computed by this model, is higher than 0.5375. There were 35 multivariate outliers detected, of which none was significant, so not excluded. The table below contains the goodness of fit of this model. As noticed earlier, for Velocity data, both completeness evaluations decrease and correctness increases.

<table>
<thead>
<tr>
<th>Overall Completeness</th>
<th>Completeness</th>
<th>Correctness</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>84%</td>
<td>86%</td>
<td>90%</td>
<td>Tomcat</td>
</tr>
<tr>
<td>74%</td>
<td>59%</td>
<td>100%</td>
<td>Velocity</td>
</tr>
</tbody>
</table>

Table 7.29: Goodness of fit of Model 1

Alternative Models constructed using Tomcat data
Next, some alternative models will be constructed with fewer variables as input to the forward selection procedure. The next variable pool which will be available for model construction contains all measures as earlier excluding CBOImp, which was the most significant variable selected in the previous model construction besides Size measures:

IC_OM, IC_OC, IC_OCC, NG_IC_OCC, IC_CM, IC_CC, IC_CCC, NG_IC_CCC, EC_CM, EC_CCC, NG_EC_CCC, CBO, NG_CBO, NG_CBOImp, NMImp and NAImp.

The following tables list the information regarding Model 2:

<table>
<thead>
<tr>
<th>Measure</th>
<th>Coefficient</th>
<th>Std. Err.</th>
<th>p-value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intercept</td>
<td>-2.7016</td>
<td>0.4733</td>
<td>0</td>
</tr>
<tr>
<td>NG_IC_CCC</td>
<td>0.0292</td>
<td>0.0099</td>
<td>0.0012</td>
</tr>
<tr>
<td>NG_CBO</td>
<td>0.0142</td>
<td>0.0058</td>
<td>0.0150</td>
</tr>
<tr>
<td>NAImp</td>
<td>0.2222</td>
<td>0.0666</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 7.30: Model 2 and its participating measures

Model 2 includes, besides the size measure NAImp, also NG_IC_CCC and NG_CBO, which represent dynamic (at the class level of measurement) and static neighbor coupling respectively. These neighbor coupling measures are derived from IC_CCC and CBO, of which both have in common to include import coupling. The log-likelihood of the model is $-61.0855$, the conditional number is $7.49$, which does not indicate any problem, and the R2 is $0.4003$. For applying this model to the 147 classes of Tomcat, a class was classified ‘predicted fault-prone’, if its predicted probability to contain a fault, computed by this model, is higher than 0.7. There were 65 multivariate outliers detected, of which none were found significant. The table below contains the goodness of fit of this model. As noticed with previous models constructed using Tomcat data, for Velocity data, both completeness evaluations decrease and correctness increases, as is the case with the size-only model.
In the next step, a model primary consisting of dynamic coupling measures and size measures was built. Therefore, those neighbor coupling measures had to be removed, which together with any dynamic coupling measure would result in a too high p-value for the model. After some inspection (forcing different subsets of measures in models and evaluate them), all dynamic neighbor coupling measures were removed from the measure-pool, together with NG_CBOImp. The current variable pool consists of the following measures:

IC_OM, IC_OC, IC_OCC, IC_CM, IC_CC, IC_CCC, EC_CM, EC_CCC, CBO, NMImp and NAImp. The resulting model, model 3, its aspects are covered by the tables below:

<table>
<thead>
<tr>
<th>Measure</th>
<th>Coefficient</th>
<th>Std. Err.</th>
<th>p-value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intercept</td>
<td>-2.2187</td>
<td>0.4057</td>
<td>0</td>
</tr>
<tr>
<td>IC_OM</td>
<td>0.0303</td>
<td>0.0148</td>
<td>0.0120</td>
</tr>
<tr>
<td>CBO</td>
<td>0.0842</td>
<td>0.0386</td>
<td>0.0233</td>
</tr>
<tr>
<td>NAImp</td>
<td>0.2385</td>
<td>0.0648</td>
<td>0</td>
</tr>
</tbody>
</table>

Model 3 includes, besides the size measure NAImp, also IC_OM and CBO, of which the later is more significant (and which also confirms univariate regression analysis). The log-likelihood of the model is **-66.8434** and the conditional number is **7.6707**, which still does not indicate any problem. The R2 value is **0.3394**, which is just a bit lower than model 2. For applying this model to the 147 classes of Tomcat, a class was classified ‘predicted fault-prone’, if its predicted probability to contain a fault, computed by this model, is higher than **0.4750**. There were 38 multivariate outliers detected, of which only 1 was found significant and excluded. The table below contains the goodness of fit of this model. As noticed with previous models constructed using Tomcat data, for Velocity data, both completeness evaluations decrease and correctness increases.
The following metrics-pool excludes CBO:

- IC_OM, IC_OC, IC_OCC, IC_CM, IC_CC, IC_CCC, EC_CM, EC_CCC, CBO, NMImp and NAImp. The last model, model 4, its aspects are covered by the tables below:

<table>
<thead>
<tr>
<th>Measure</th>
<th>Coefficient</th>
<th>Std. Err.</th>
<th>p-value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intercept</td>
<td>-1.8737</td>
<td>0.3540</td>
<td>0</td>
</tr>
<tr>
<td>IC_OM</td>
<td>0.0366</td>
<td>0.0149</td>
<td>0.0025</td>
</tr>
<tr>
<td>NAImp</td>
<td>0.2737</td>
<td>0.0623</td>
<td>0</td>
</tr>
</tbody>
</table>

Model 4 includes, besides the size measure NAImp, the dynamic coupling measure IC_OM. The log-likelihood of the model is \(-69.6541\), the conditional number is \(5.3975\), which does not indicate any problem, and the R2 is \(0.3162\). For applying this model to the 147 classes of Tomcat, a class was classified ‘predicted fault-prone’, if its predicted probability to contain a fault, computed by this model, is higher than \(0.3250\). There were 29 multivariate outliers detected, of which none was found significant. The table below contains the goodness of fit of this model. As noticed with previous models constructed using Tomcat data, for Velocity data, both completeness evaluations decrease. But in this case, correctness stays the same for both data.

<table>
<thead>
<tr>
<th>Overall Completeness</th>
<th>Completeness</th>
<th>Correctness</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>80%</td>
<td>96%</td>
<td>76%</td>
<td>Tomcat</td>
</tr>
<tr>
<td>74%</td>
<td>69%</td>
<td>81%</td>
<td>Velocity</td>
</tr>
</tbody>
</table>

Table 7.34: Model 4 and its participating measures

Table 7.35: Goodness of fit of Model 4
Having built some models using Tomcat data, most of these models their overall completeness and completeness when analyzing Velocity data was less than when analyzing Tomcat data. However, the correctness was higher. This implies that for Velocity data, these models misidentify more classes to be faulty or not-faulty, but on the other hand, when a class is identified as faulty, the chance that it is indeed faulty is higher when comparing the results of these models with Tomcat data. Also, because of a lower completeness, less of the total faults in the system are found when analyzing Velocity data with these models.

7.6 Analyzing regression results

After having built so many models, an evaluation has to follow to actually derive some conclusions. Several models have been built using the Velocity data and the Tomcat data. Models from both data have shown that NAImp is a very good size predictor of fault-proneness. But as stated earlier (chapter 5), we want to improve prediction models by extending them with design metrics.

Models built using Velocity data

Those models built using the Velocity data showed that import dynamic coupling, with the class level as entity of measurement, have in univariate regression analysis the highest R2 value among the non-size measures. Only their coefficients were somewhat lower than those of the static coupling measures, but coefficients of multivariate non-linear models are difficult to interpret. Dynamic export coupling does not seem to have a strong relation with fault-proneness. In Univariate regression most export coupling measures were found insignificant, and the other dynamic export coupling measures which were found significant related to fault-proneness were not included in any model built using multivariate regression analysis. When using these multivariate models constructed using Velocity data to predict faulty classes for the Tomcat data, it is found that the overall completeness and correctness decrease while completeness increases. So for the Tomcat data, with respect to the Velocity data, more faults (in percentage of the whole) are covered by these models (where multiple fault may be from the same class), which increases the completeness, but more non-faulty classes are classified as faulty, which decreases the overall completeness and correctness of the models.
Model 3 (table 7.20: IC_OC, IC_CC and NAImp) has almost the highest R2 value, 0.4236, but the best goodness of fit compared with the other models. On the other hand, its cumulative goodness of fit regarding the Tomcat data decreases with 17 percent. Compared with model 1 (table 7.16: CBOImp and NAImp), which has a somewhat less R2 value of 0.3844, the cumulative goodness of fit of the Tomcat data increases only with 1 percent compared with the cumulative goodness of fit of the Velocity data. So, among these different data, model 1 is more stable than model 3. In this case, considering the models constructed using Velocity data, model 1 would be preferred over model 3, because it is almost as good in prediction of fault-proneness as models 3, but much more stable across the two different data studied in this thesis.

Univariate regression analysis results did not regard neighbor coupling as a very good/strong predictor of the fault-proneness of a class. Still, there was a model constructed using multivariate regression, which included NAImp together with NG_CBOImp (model 5, table 7.24). Comparing the size-only model with this model, its R2 increased from 0.2597 to 0.3122. Considering the cumulative goodness of fit regarding Velocity data, there was an increase of 14 percent, and for the Tomcat data an increase of 3 percent. Compared to the other measures (IC_OC, IC_CC and CBOImp) this is not that much, but still shows that NG_CBOImp can increase the goodness of fit of size-only models.

**Models built using Tomcat data**

The models built using **Tomcat** data showed in univariate regression that all dynamic import coupling measures are related to fault-proneness, where those measures with the class level as entity of measurement were more significant related compared with those with the object level of measurement. These models also confirm that dynamic export coupling is not a good predictor of faulty classes. What univariate regression analysis also shows here, is that static coupling measures, CBO and mostly CBOImp, were much stronger related to fault-proneness than dynamic coupling measures. CBOImp has an around twice as large R2 value and coefficients as the dynamic import coupling measures (e.g. IC_CC). This was not the case when examining the Velocity data, and has some influence in multivariate regression analysis, where the best models consist besides of size (NAImp) also of the static import coupling measure CBOImp (model 1, table 7.28). Another different observation with univariate regression analysis was that where previously NG_CBO was the only neighbor coupling measure which had a good R2 in predicting faulty classes, here all dynamic import neighbor coupling measures and the static neighbor
coupling measures showed a much stronger relation. This resulted in model 2 (table 7.30) which uses only neighbor coupling measures and NAImp.

When using these from Tomcat derived models to predict faulty classes for the Velocity data, it is found that both completeness attributes of these models decrease, while correctness increases. This implies that for the Velocity data, these models miss-identify more classes compared to the Tomcat data and less faults are identified (where multiple faults may be of the same class), but on the other hand, when a class is identified as faulty, the chance that it is indeed faulty is higher when compared with the results regarding the Tomcat data.

Model 1 (table 7.28: CBOImp and NAImp) has the highest R2 value, 0.4491, and the best goodness of fit compared with the other models. Its cumulative goodness of fit regarding the Velocity data decreases with 17 percent, which is not that stable as model 1 derived from Velocity data (table 7.16: CBOImp and NAImp), which includes the same measures. The next model which is almost as stable is model 4 (table 7.34: IC_OM), containing the dynamic import coupling measure IC_OM. However, its R2 value (0.3162) and its goodness of fit decrease much compared with model 1(table 7.28: CBOImp and NAImp).

Where neighbor coupling showed not a promising predictive power toward the fault-proneness of a class when analyzing Velocity data, here the second multivariate model consists of only neighbor coupling measures (model 2, table 7.30: NG_IC_CCC and NG_CBO and NAImp) besides NAImp. The R2 value of 0.4003 is also very close to the best model, model 1(table 7.28: CBOImp and NAImp). The only downside, and because of which this model is not considered appropriate, is that its cumulative goodness of fit regarding the Velocity data decreases with 32 percent. So, among both data, Velocity and Tomcat, this model is not that stable.

**Differences in Velocity and Tomcat data**

Besides **differences** in dynamic import coupling, neighbor coupling and static coupling measures between analysis of velocity data and Tomcat data, models derived from velocity data overall have a somewhat better goodness of fit for both data, Velocity and Tomcat, than models derived from Tomcat data, and are therefore more stable. There are two possibilities of how these differences are caused, a change in the correlation of some measures with size, and a difference of
usage of object-oriented features like polymorphism and dynamic binding between the two data collections.

The large increase of the strength of CBOImp (and also somewhat regarding CBO) as a predictor of the fault-proneness of a class may be explained by its increased correlation with the size measures NMImp and NAImp. For CBOImp, its Spearman’s rho value for NMImp increased from 0.2880 when analyzing Velocity data to 0.5138 when analyzing Tomcat data. For NAImp the increase was from 0.1241 to 0.5233, which is more than 400%. Keeping in mind that both these size measures show a strong relation towards fault-proneness of a class in univariate regression upon the Tomcat data, and that CBOImp has a much increased correlation with these size measures when comparing the results on Velocity data and Tomcat data respectively, it might not be a surprise that CBOImp’s predictive power of a class its faults has increased.

Another possible way to explain these differences is examining the difference in the DII in both data collections.

Having defined DII in section 3.4, it will be used upon Velocity and Tomcat data in order to explain the differences in the previous data analysis. The collected data is listed in the table below for both data collections:

<table>
<thead>
<tr>
<th></th>
<th>Velocity</th>
<th>Tomcat</th>
</tr>
</thead>
<tbody>
<tr>
<td>Messages</td>
<td>765050</td>
<td>302532</td>
</tr>
<tr>
<td>Distinct Invocations</td>
<td>1861</td>
<td>3033</td>
</tr>
<tr>
<td>Distinct occurrences</td>
<td>54</td>
<td>40</td>
</tr>
<tr>
<td></td>
<td>2.9</td>
<td>1.32</td>
</tr>
</tbody>
</table>

Table 7.36: Object Orientation analysis

As can be seen from the table, the distinct occurrences of invocations of methods from super classes upon a class do not differ much between Velocity and Tomcat. 54 versus 40. But as explained earlier in section 3.4, the relation with respect to the number of distinct invocations
should be taken, which differ very much, \( \frac{3033}{1861} \times 100\% = 163\% \). Using the number of messages instead would not be valid because a smaller system can generate more messages than a bigger one. The resulting DII’s, where

\[
\text{DII} = \left( \frac{\text{Distinct occurrences of invocation thru inheritance}}{\text{Distinct Invocations}} \right) \times 100
\]

differ very much. The DII of Velocity is more than twice as large compared to the DII of Tomcat:

\[
\frac{2.9}{1.32} = 2.2
\]

Interpreting this metric as an indicator for usage of object-oriented features, the derived values state that Velocity makes more than twice as much usage of object-oriented features compared to Tomcat. This may explain how the relation between dynamic import coupling and static coupling towards faulty classes differ in Velocity and Tomcat.

When no or very little use is made of object-oriented features like inheritance and polymorphism, it is very likely that for a message, the invoked method is declared in the class type of the object that message was send to. This would result in the derived invocation containing that class, which in turn should result in the dynamic coupling measures with the object level of entity (the IC_Ox measures) not to be much different from the dynamic coupling measures with the class level of measurement. Both types of measures would have almost the same underlying data variance, which should be made visible in PCA as both types of measures would share the same PC.

This was the case for the Tomcat data, and supports the explanation given above for the differences found among the results of Velocity and Tomcat data.

### 7.7 Hypotheses evaluation

The hypotheses made in chapter 5 will be evaluated in this section considering the previous data analysis:

- **H-IC** - The regression analysis on both data show that import coupling is significant related to the fault-proneness of a class. Therefore this hypothesis is considered approved.
- **H-EC** - Export coupling showed to be a very bad predictor of the fault-proneness of a class. None of the export (neighbor) coupling measures were part of any prediction model. Therefore this hypothesis is rejected.

- **H-IC-ENV** - Import neighbor coupling measures showed to be significant predictors of fault-proneness. When analyzing the Tomcat data, NG_IC_CCC had even a bigger R2 value than any dynamic import coupling measure. Also, using both data, models were constructed using import neighbor coupling measures. Model 5, table 7.24 uses NG_CBOImp with NAImp and model 2, table 7.30 uses NG_IC_CCC together with NG_CBO and NAImp. These models showed that including these measures led to an increase in the goodness of fit compared to size-only measures, which is why this hypothesis is considered approved. On the other side, models were constructed using non-neighbor coupling measures, which had a better goodness of fit. Neighbor coupling measures therefore still have to prove themselves to be useful (maybe in other areas of software metrics than fault-proneness prediction).

- **H-EC-ENV** - From univariate regression analysis it showed that export neighbor coupling measures were not significant predictors of fault-proneness. They were therefore also excluded in further data analysis. This hypothesis is rejected.

- **H-DC** - According to the DII metric, in Tomcat little usage was made of object-oriented features compared to Velocity. This was confirmed by the PCA analysis upon Tomcat data where all dynamic import coupling measures were sharing the same pc. There was no distinct pc for dynamic import coupling measures with the class and object level as the entity of measurement. Univariate regression in section 7.4 showed that some import coupling measures have a higher R2 value than the CBO and CBOImp measures when using Velocity data. This was not the case when using Tomcat data. From all this, we can conclude that more usage of object-oriented features will lead to dynamic import coupling measures being more predictive with respect to the fault-proneness of a class. Therefore, this hypothesis is considered approved.

Further, one could notice that regarding both data collections:

1. Dynamic import coupling measures with the class level as entity of measurement are better predictors of fault-proneness than the same measures with the object level as
entity of measurement. For an overview, see table 7.10 and 7.12 and Univariate regression results (section 7.4).

2. Dynamic import coupling measures with distinct methods as the level of strength are according to both data collections evaluated better predictors of fault-proneness than those measures with the distinct classes as strength level. In the same way, these measures with distinct classes as level of strength are better predictors of fault-proneness than those with the “class-to-class” level of strength (see the Univariate regression results in section 7.4).

It is obvious that collecting dynamic metrics demands more efforts than collecting static ones. This is acceptable if these dynamic metrics are significant better predictors of external attributes (fault-proneness). In this thesis we have seen, that if relative much usage is made of object-oriented features like inheritance, polymorphism, dynamic binding etc., dynamic coupling metrics are better predictors of fault-proneness. However, if models using these metrics are used upon data where not much usage is made of these object-oriented features, the extra effort is not justified, because dynamic metrics will not be that effective. Also, one has to evaluate whether a software system makes “enough” usage of object-oriented features for dynamic coupling measures being significant. “Enough” has to be defined, which is left for future investigation (one may define a minimum DII value). If a model is needed for usage across different environments where enough usage of object-orientated features is not certain, perhaps static coupling metrics like CBOImp stand a better chance for stable results or maybe in combination with some dynamic coupling measures.
Chapter 8

Conclusions and Future work

Analyzing the models constructed using Velocity data, where dynamic import coupling measures with the class level of measurement were found stronger related to fault-proneness than static coupling, the model containing IC_OM, IC_CC and NAImp turned out to be the best predictive
measures of fault-proneness. But taking into account stability of prediction model across different
data, the best model would be the models containing CBOImp and NAImp, which considering its
goodness-of-fit was almost as good as the previous model. When Analyzing Tomcat data, Static
coupling measures were found to be much stronger related to fault-proneness than when studying
Velocity data. With the results of multivariate analysis on in mind, the best and most stable model
consisted of the measures CBOImp and NAImp.

Still, the predictive power of dynamic import coupling measures is high when analyzing Velocity
data. The problem is that for some software, like Tomcat, which make less usage of object-orientated features, they may be less predictive, which in turn lowers their stability as a prediction model. This is not the case with static coupling measures. To incorporate these differences between software systems, it wouldn’t be a strange idea, when one tries to build a prediction models with great stability across different software system, to besides size measures, make usage of static import coupling measures for stability, and dynamic import coupling measures for there good predictive power in some software systems. In this study, it was not possible to create a multivariate model containing besides size and CBOImp, also one or more dynamic import coupling measures. This because the p-values would increase over the maximum allowed value of 0.05. Future work therefore includes analyzing dynamic coupling measures together with other static coupling measures like the measures of the suite by Briand et al. which may lead to a model consisting of an allowed combination of static and dynamic coupling measures, which besides stable across different software systems, may also be a very strong predictor of external software attributes (fault-proneness).

Export coupling measures were not found to be significant predictors. This also holds for export neighbor coupling. For the dynamic import coupling measures, it was found measures with the class level as entity of measurement are better predictors of fault-proneness than the same measures with the object level as entity of measurement, and measures with distinct methods as the level of strength are better predictors of fault-proneness than those measures with the distinct classes as strength level, which in the same way are better predictors of fault-proneness than those with the “class-to-class” level of strength.

The import neighbor coupling measures showed good results when analyzing Tomcat data. Their goodness of fit was almost as good as the best model constructed, but their stability was not that good. So these measures, like the dynamic measures, suffer from the same problem, differences
among software system. In the future, these measures should be investigated together with other measures excluded in this thesis.

To better understand why for certain software systems some measures give unexpected results, an investigation should be done in order to find out what the differences between software systems can be, and how they are related to software metrics. This will give us a better understanding about the predictive power of measures among different software environments. In the last section, the usage of invocations of methods of superclasses was analyzed, which turned to be a possible explanation for differences among the relation between the dynamic measures used and fault-proneness, but which is just a part of the whole picture. The used DII metric should also be extended with more object-oriented features/information like e.g. dynamic binding, which is also left for future work.

---

BIBLIOGRAPHY


27. S. Franklin, M. Brodeur. A Practical Application of a Robust Multivariate Outlier
   Detection Method, Statistics Canada.
   Estimates and Confidence Intervals for Mahalanobis Distance, paper 163-30.
34. T. Kamiya, S. Kusumoto, K. Inoue. Prediction of fault-proneness at Early Phase in
   Object-Oriented Development. Graduate School of Engineering Science, Osaka
   University.
36. W.J. Krzanowski, F.H.C. Marriott. Multivariate analysis: Classification, covariance
39. J. Lindroos. Code and Design Metrics for Object-Oriented Systems. Department of
41. W. Mendenhall, J.E. Reimuth, R.J. Beaver. Statistics for Management and Economics,
42. M. Nijsse. Testing the Significance of Kendall’s t and Spearman’s rS.
   Department of Computer Science and Electrical Engineering, West Virginia University.
Appendix A

The design of Jrev

In this appendix, the design of a software application/tool, Jrev, will be explained. For this project, measures were needed to be taken from two Java software systems (Velocity and Tomcat), which will be explained later on in this section. The needed measures are those dynamic coupling measures explained earlier and also some static coupling metrics and size metrics. In order to accomplish this, a software system had to be developed, whose design will be explained in this section.
A.1 Introduction to Jrev

Purpose of the System

The purpose of Jrev is to collect run-time information of the input-program, which is given as input to Jrev. While running this input-program on JVM, Jrev has to collect enough information, in order to compute all the dynamic coupling measures explained earlier in this thesis. Besides this functionality, Jrev also has to be able to, given a Java jar file, collect static data/measures for all the classes within the jar file.

Objectives of the project

The objective for developing Jrev, are to collect the explained and selected dynamic and static coupling measures of two selected software systems, Velocity and Tomcat, which will be used for further analysis in this project.

A.2 Proposed System

In this sub-section, the proposed system which has to capture the needed functionality is explained. First all the different types of requirements will be given, followed by the design models to accomplish these requirements. At last, the API for 3rd party software will be given, which has to make it possible to derive the collected data from Jrev for further usage e.g. another application which computes other dynamic measures using these collections of data made available by Jrev thru this API.

The functionality of Jrev can be divided in two parts. The first part, which I will refer to as the dynamic part of Jrev, takes as input a Java application’s main function [6 and 44]. While running this program, Jrev has to trace and log all the classes of this input-program loaded by the JVM during execution, along with all the objects created, the methods of these classes which are invoked at some point and all the messages generated by these objects while executing this input application. The functionality of the JDI library is used to accomplish this. Depending on the user input parameters, the system should output these findings in some textual form or in some graphical way. All these collections of data logged by Jrev during execution of the input application have to be made available for 3rd party software systems for further usage, or in the same way for Jrev itself to compute from these data collection the dynamic coupling measures of the input-application. The second part of Jrev, which I will refer to as the static part of Jrev, acts like as a standalone application (which the first part does also), and takes as input a Java jar file. The purpose is to derive for each class, if not filtered with respect to the users input, some static measures as will be explained later on. Using this information as a base, the different types of requirements will be extracted and explained more specifically. The functionality of the Bloat library I used to extract static information from java code (jar or class files).

A.2.1 Functional requirements

For the dynamic part of Jrev, the following functional requirements are put:
Input file

The input file consists of a path to a java “.class” file, which represents the input-application of which Jrev will collect run-time data. The Java programming language compiles java source code to byte code instructions, which, when executed, will be interpreted by a JVM. A plus point of this method is that java programs are platform independent. You only need a JVM for the platform you are using.

The input file to this system, a “.class” java file, is thus a java byte code file, which can be interpreted by a JVM. The input file has another requirement. It has to contain a main method, which by the Java specification should look like:

“public static void main(String[] args)”

Jrev will execute this main function of the input-application. This input application is considered to exercise some Java application of which Jrev will collect run-time data. The more functionality this input-application exercises the Java application it executes (coverage), the more realistic the output of Jrev will be regarding dynamic metrics.

Tracing the execution of the input file

Jrev initializes a JVM thru the JDI library and runs the input “.class” file, as just explained, on that JVM. The JVM will interpret the input file by invoking the main method within the input class file. Using the JDI (Java Debugger interface) Jrev logs events from the JVM to trace the execution of the input program. From this retrieved information data collections (classes, methods, objects and messages) will be generated.

Multiple threads

Jrev has to be able to handle input-applications which consist of multiple threads of execution. Also, Jrev has to be able to distinguish between messages of on thread and those of another thread.

Classes

Jrev has to collect all the java classes loaded by the JVM when executing/tracing the input program. Because this collection can be very big, Jrev should support some filtering mechanism so that the user can be able to filter certain classes e.g. filter java classes of the java.lang.* package.

For each class collected, its full name (including package name of that class) should be determined. In the output, all the class names of the class-collection should be listed in some text file of which the user can give the name as an input-parameter to Jrev. Otherwise a standard text file in the current working directory will be used e.g. classes.txt. For each class the stored information syntax supplied by Jrev is: packagename.classname
Methods

Jrev has to collect all the methods of the non-filtered classes collected, which are invoked while executing/tracing the input-application. For each invoked method, the name, the return type, the formal parameters and the enclosing class name (including its package name) has to be determined. If the class within which a method is declared is listed to be filtered, than also this method has to be filtered.

This collection of methods includes different kinds of methods: public, private, protected and static methods. In the output, all the names of the method-collection (including there full class names) should be listed in some text file of which the user can give the name as an input-parameter to Jrev. Otherwise a standard text file in the current working directory will be used e.g. methods.txt. For each method the stored information syntax supplied by Jrev is:

```
packagename.classname.methodname
```

Objects

Jrev has to collect all the objects initiated while executing the input-application. For each object, besides its type, which includes a class name preceded by a package name, a unique identifier is needed to make a distinction between two different objects of the same type possible. In the output, all the names of these objects (including there full class names and there unique identifier) should be listed in some text file of which the user can give the name as an input-parameter to Jrev. Otherwise a standard text file in the current working directory will be used e.g. objects.txt. For each object name the stored information syntax supplied by Jrev is:

```
packagename.classname:objectid
```

Messages

Jrev has to log all the messages during the execution of the input-application.

A message is a signal from a source to a destination objects. Within a source object, the message is initiated from a method. This source-method cannot be a static method, because it is initiated by an object. The message is send to another method of an object. For a better overview, a message consists of:

1. the method from which this message was send, referred to as “the from-method”
2. the object from which the from-method was triggered, referred to as “the from-object”
3. the object to which the message is send, referred to as “the to-object”
4. the method of the to-object to which the message was send, referred to as “the to-method”

In the output, all the information (as listed) of these messages should be listed in some text file of which the user can give the name as an input-parameter to Jrev. Otherwise a standard text file in the current working directory will be used e.g. msg.txt. For each message, the stored information syntax supplied by Jrev is:

```
packagename.classname:objectid. packagename.classname.methodname
```

Method Invokations
Using the collection of Messages, Jrev computes the collection of Method invocations as per the algorithm of chapter 2. For each invocation the stored information syntax supplied by DCoupling is:

\[\text{packagename.classname.methodname} \rightarrow \text{packagename.classname.methodname}\]

**Output**

The total output consists of:

1. A class file, which consist all the class names including there package name. This collection excludes those classes the user wanted to be filtered (see User-Input below).
2. A methods file, which consists of all the methods of the classes listed in the class-file, which were invoked during execution of the input-application.
3. An objects file, which consists of all the objects of the classes listed in the class-file, which were initiated during execution of the input-application.
4. A message file, which consists of all the messages send in the system, while executing the input-application. The class types of both the objects in each message, have to be part of the class file, otherwise the message will be excluded.
5. For each thread, in a separate directory, a text file will be written containing the sequence of the messages in that thread. Default directory name within the current working directory is seq_output_dir. The different files will be name by default “seq_x.txt”, where x is substituted by the number of the thread. An additional file will be written, seq_all.txt, containing the sequences of all the message of all the threads.
6. if the ‘-ho’ option is used, then for each collection file ( classes, methods, objects, and messages), an additional file will be written, with a ‘2’ appended to the original name, and where the output consists of the same information, only, the layout of the output will be horizontal instead of the default vertical output.
7. if the ‘-dc filename’ option is used, then Jrev will calculate all the dynamic coupling metrics explained in chapter 2 and also the ones defined in chapter 3 ( including the neighbor coupling measures) for each class of the input-application. This will be written to a text file with the name “filename.txt”. If the ‘-ho’ option is also used, another file, “filename2.txt” will be output containing the same information as prior, but with a horizontal output layout. In addition to these files, another file, “dcstats.txt”, will be output, containing the method invocations derived from the collection of messages.

**Usage as a Library thru an API and Extensibility**

3\textsuperscript{rd} party software should be able to use Jrev its functionality as a library. To make this possible, Jrev has to make some functionality available to the outside (public interface). These functions should contain the following:

- Supply Jrev a path to an input-application of which It has to collect run-time data.
- Retrieve the collections computed by Jrev for further usage outside the scope of Jrev.

This public API (Application Programming Interface) to use Jrev its functionality within other programs is listed below in section 4.3.2.5. Off course, it should be possible to give Jrev other parameters, and let it output data file for the classes, methods, objects, messages and dynamic coupling, which in turn can be used by another program as input.
Because one part of Jrev is designed to collect run-time data of the input-application in some data structures, and make these data collections available to other parts of Jrev as well to 3rd party applications, it is very easy to extend Jrev with new functionality, which uses these data collections in order to provide some addition functionality e.g. compute with the ‘-dc’ option the dynamic coupling measures from these data collections. It is also easy to extend Jrev with another (dynamic) measure which uses these data collections. It should also be possible, to easily extend Jrev with other functionality like an analysis tool to analyze this data, outputting this data in XML format etc.

For the static part of Jrev, the following functional requirements are put:

Input file

The input Jrev consists of a path to a java Jar file. Of all the classes within this Jar file, excluding those who are filtered thru the ‘-filter’ option, static information will be derived by using the bloat library, and some static measures will be computed, CBO, NG_CBO, CBOImp, NG_CBOImp, CBOExp, NG_CBOExp, NMImp and NAImp, which will be written to an output file.

Output

The total output consists of a text file containing the names of the analyzed classes, together with there values of the computed measures.

A.2.2 Non-functional requirements

User Interface

Jrev will be a command-line application. From the command line, the user has to execute Jrev with the preferred parameters. Jrev should be able to output all computed collection in a readable textual format. Which output file to be used and which data collection(s) to output should be specified by the user by using parameters, otherwise default collections with default names will be used.

In order to execute the dynamic part of Jrev, the following code has to be used:

```
java -classpath %CLASSPATH%;<path>  jrev.Jrev <options>
```

Where <path> is the path to where Jrev is installed and <options> are the supplied parameters to Jrev as explained below.

User-Input

The following user input/parameters to Jrev are possible:
(Input parameters followed by the * token are not optional)

- **main**
This parameter is followed by the path to a java class file, which will be regarded as the input-application, and has to contain a main function [6 and 44]. Jrev will execute this application, and trace it to collect data.

e.g. “-main c:\example\Example”, where in the c:\example directory a file “Example.class” should be, which contains a Java Main method.

-cp
Allows the user to supply all the necessary paths to Java packages (jar files), which are needed by the input-application in order to run/execute. If the input-application needs a certain library which is not in the class-path of the operating-system, and it is not supplied thru this option, then Jrev will eventually throw an exception and terminate.

-filter
Allows the user to provide a string of text to Jrev consisting of class names (including their package name) which Jrev has to filter. The ‘;’ token is used to separate different names. It is possible to filter whole packages by using the ‘*’ token to represent all the classes of the named package.

e.g. “-filter org.apache.catalina.tester;org.apache.startup.*;java.*;javax.*”

-class
Allows the user to give the path of a text file which Jrev will use to write the collection of classes to. If not supplied, the default file, class.txt, will be used (in the current working directory).

-meth
Allows the user to give the path of a text file which Jrev will use to write the collection of methods to. If not supplied, the default file, method.txt, will be used (in the current working directory).

-obj
Allows the user to give the path of a text file which Jrev will use to write the collection of objects to. If not supplied, the default file, obj.txt, will be used (in the current working directory).

-msg
Allows the user to give the path of a text file which Jrev will use to write the collection of messages to. If not supplied, the default file, msg.txt, will be used (in the current working directory).

-show
This option will let Jrev display all the generated sequence files (see the –seq option) using the Sequence program of Alex Moffat (http://www.zanthan.com/itymbi/). The sequence files will be displayed like a UML sequence diagram.

-png
This option will let Jrev generate png files for all the generated sequence files (see the –seq option) using the Sequence program of Alex Moffat (http://www.zanthan.com/itymbi/). The png files will show the sequences like UML sequence diagrams.

-seq
Allows the user to provide a directory name, in which all the sequence files will be written. If not supplied, a default directory within the current directory ill be used, named “seq_output_dir”.

137
Sequence files for all threads will be generated along with an additional file containing information regarding all threads.

**-dc**
This option informs Jrev to compute from the collected data, all the dynamic coupling measures described in this thesis. The output layout is vertical.

**-ho**
This option is only to be used in conjunction with the –dc option. By using this option, Jrev will output an additional file containing for each class the dynamic coupling measures, but the output layout is horizontal. The standard file with the same dynamic coupling measures written using a vertical layout is still written.

**-help**
This option will print all the possible user-input on screen, and terminate Jrev immediately.

**Memory usage**
In order to support tracing of larger java applications, Jrev has to consume as less memory as possible. This, because otherwise there will be a chance that the system on which Jrev will be running will run out of memory when tracing large input-applications. With large systems I mean systems which consist of large collections of objects and messages compared to the available memory of the JVM. Therefore Jrev should store per data element as less as information as possible.

In order to execute the static part of Jrev, the following code has to be used:

```
"java -cp <path1> jrev.util.metrics.Metrics <path2> <options>"
```

where path1 refers to the path of the Jrev installation, and path2 to the libraries needed by the input program.

For the static part of Jrev, the following options are available:

**User-Input**

The following user input/parameters to Jrev is possible:
(Input parameters followed by a ‘*’ token are not optional)

**file-name** *
A path to a Java jar file which will be used by Jrev for further analysis.

**-nointerf**
This option will let Jrev filter out classes which are interfaces.
-filter classnames
This option gives the user the possibility to filter out some classes or packages within the supplied Java Jar file.
The ‘;’ token is used to separate different names. It is possible to filter whole packages by using the ‘*’ token to represent all the classes of the named package.
e.g. “-filter org.apache.catalina.tester;org.apache.startup.*;java.*;javax.*”

-order filename
Here, the user can supply a text file containing the class names he is interested in, in some order. Jrev will output the measures in some text file, see the option ‘-metrics’, according to the order of classes in filename. Usually the class output file of the dynamic part of Jrev is used with this option.

-cbo
Using this option, Jrev will output in the output file, see the option ‘-metrics’, the static coupling measures: CBO, NG_CBO, CBOImp, NG_CBOImp, CBOExp and NG_CBOExp.

-sizem
Using this option, Jrev will output in the output file, see the option ‘-metrics’, the size measures: NMImp and NAImp.

-metrics filename
Jrev will use filename as the output file to write the chosen metrics by the user. If this option is not used, the default file “metrics.txt” will be used in the current directory. An additional file with the same name, but a ‘2’ token appended to it, will be output using a horizontal layout, containing the same data.

A.2.3 Pseudo requirements

Java
Jrev will be implemented in the java programming language. This leads to a platform independent executable.

Java Debugger Interface (JDI)
Jrev will be using the JDI (Java Debugger Interface) library in order to trace a program running on a JVM.

Bloat
Jrev will be using the Bloat library in order to extract static information from Java class files.

A.2.4 System models
In this part, the design of the whole Jrev System will be explained.

Use case model

Figure 4.2 depicts the UML use case diagram of the dynamic part of Jrev. **Actors** in this system are:

1. **User**
   This is the normal user, person, who wants to execute Jrev.

2. **3rd Party Software**
   This actor represents other software systems, which use Jrev thru the Jrev its public API to use its functionality.

Below a description is given of each use cases.
Figure 4.2: UML Use case diagram of dynamic part of JRev
Use case name: ExecutionProgram

Participating actor: User

Entry condition: This use case uses the CheckInput, TraceProgram and WriteOutput use cases. The system, Jrev, is not running yet, or is terminated from a previous execution.

Flow of events:

1. The User writes the command he wants to execute on the command line with respect to the options explained previously. He presses “Enter” on the keyboard of the pc.
2. Initiate the CheckInput use case.
3. Initiate the TraceProgram use case.
4. If the –dc option was used by the user; the DCcoupling use case will be initiated.
5. Initiate the WriteOutput program.

Exit condition: none

Special requirements: In the command used in event ‘1’, a path to a .class file should be given as a parameter, and the Java class this file represents should contain a main method.

Use case name: CheckInput

Participating actor: none

Entry condition: User should have started the ExecuteProgram use case, and the system, Jrev, should not be running yet. This use case initiates the Help use case if the user input is not conform the specification.

Flow of events:

1. check if the –main option is used. If not, initiate the Help use case.
2. if the –seq option is used, check whether the next user parameter is a path a directory. If the next parameter to ‘-seq’ is not a path to a directory, initiate the Help use case.
3. if the –class option is used, check whether the next user parameter is a path to a file. If the next parameter is not a path to a file name, init the Help use case.
4. if the –obj option is used, check whether the next user parameter is a path to a file. If the next parameter is not a path to a file name, init the **Help** use case.

5. if the –method option is used, check whether the next user parameter is a path to a file. If the next parameter is not a path to a file name, init the **Help** use case.

<table>
<thead>
<tr>
<th>Exit condition</th>
<th>none</th>
</tr>
</thead>
<tbody>
<tr>
<td>Special requirements</td>
<td>none</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Use case name</th>
<th>Help</th>
</tr>
</thead>
<tbody>
<tr>
<td>Participating actor</td>
<td>none</td>
</tr>
<tr>
<td>Entry condition</td>
<td>Incorrect usage of input parameters.</td>
</tr>
</tbody>
</table>
| Flow of events      | 1. Print the usage of Jrev out on the screen.  
2. Exit the system. |
| Exit condition      | none                                      |
| Special requirements| none                                      |

<table>
<thead>
<tr>
<th>Use case name</th>
<th>TraceProgam</th>
</tr>
</thead>
<tbody>
<tr>
<td>Participating actor</td>
<td>none or 3rd Party Software</td>
</tr>
<tr>
<td>Entry condition</td>
<td>input parameters supplied by User should be first tested by the <strong>CheckInput</strong> use case. Uses the <strong>Help</strong> use case if the supplied class file does not exist or is not a valid compiled java file or does not contain a valid main function for execution.</td>
</tr>
</tbody>
</table>
| Flow of events      | 1. Start execution of the input program’s main function, and trace it. If the input class file is not valid, initiate the **Help** use case.  
2. Store events from the JVM for all classes initiated by the JVM, the methods invoked, objects initiated and messages send. This collected information excludes information regarding filtered classes. |
<p>| Exit condition      | none                                      |
| Special requirements| none                                      |</p>
<table>
<thead>
<tr>
<th>Use case name</th>
<th><strong>DCcoupling</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Participating actor</td>
<td>none or 3rd Party Software</td>
</tr>
<tr>
<td>Entry condition</td>
<td>The <strong>TraceProgam</strong> use case should be run prior to this use case.</td>
</tr>
</tbody>
</table>
| Flow of events      | 1. get the collection of classes thru the getClasses use case.  
                      2. get the collection of methods thru the getMethods use case.  
                      3. get the collection of objects thru the getObjects use case.  
                      4. get the collection of messages thru the getMessages use case.  
                      5. initiate the **Compute Invocations** use case.  
                      6. get the collection of invocations thru the **getInvocations** use case.  
                      7. initiate the **Compute Dynamic Coupling** use case.  
                      8. get the coupling data thru the **getCouplingResults** use case. |
| Exit condition      | Dynamic coupling measures data should be available. |
| Special requirements| none                   |

<table>
<thead>
<tr>
<th>Use case name</th>
<th><strong>WriteOutput</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Participating actor</td>
<td>none or 3rd Party Software</td>
</tr>
<tr>
<td>Entry condition</td>
<td>The <strong>TraceProgam</strong> use case should be run prior to this use case.</td>
</tr>
</tbody>
</table>
| Flow of events      | 1. if the –seq option was used by the User, than write the collection of sequences, as explained in the specification, in a textual form in the supplied (or default) directory. If the –sep option was used, than each thread-sequence will be written to a separate file.  
                      2. if the –class option was used by the User, than write the Class collection in a textual form to the specified file.  
                      3. if the –obj option was used by the User, than write the Object collection in a textual form to the specified file.  
                      4. if the –method option was used by the User, than write the Method collection in a textual form to the specified file.  
                      5. if the –msg option was used by the User, than write the Message collection in a textual form to the specified file. |
<p>| Exit condition      | none                   |
| Special requirements| none                   |</p>
<table>
<thead>
<tr>
<th>Use case name</th>
<th>getClasses</th>
</tr>
</thead>
<tbody>
<tr>
<td>Participating actor</td>
<td>none or 3rd Party Software</td>
</tr>
<tr>
<td>Entry condition</td>
<td>can be initiated only after the TraceProgram use case has been initiated.</td>
</tr>
<tr>
<td>Flow of events</td>
<td>Return the collection of classes, stored in the last run of the TraceProgram use case, for further usage.</td>
</tr>
<tr>
<td>Exit condition</td>
<td>none</td>
</tr>
<tr>
<td>Special requirements</td>
<td>none</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Use case name</th>
<th>getMethods</th>
</tr>
</thead>
<tbody>
<tr>
<td>Participating actor</td>
<td>none or 3rd Party Software</td>
</tr>
<tr>
<td>Entry condition</td>
<td>can be initiated only after the TraceProgram use case has been initiated.</td>
</tr>
<tr>
<td>Flow of events</td>
<td>Return the collection of methods, stored in the last run of the TraceProgram use case, for further usage.</td>
</tr>
<tr>
<td>Exit condition</td>
<td>none</td>
</tr>
<tr>
<td>Special requirements</td>
<td>none</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Use case name</th>
<th>getObjects</th>
</tr>
</thead>
<tbody>
<tr>
<td>Participating actor</td>
<td>none or 3rd Party Software</td>
</tr>
<tr>
<td>Entry condition</td>
<td>can be initiated only after the TraceProgram use case has been initiated.</td>
</tr>
<tr>
<td>Flow of events</td>
<td>Return the collection of objects, stored in the last run of the TraceProgram use case, for further usage.</td>
</tr>
<tr>
<td>Exit condition</td>
<td>none</td>
</tr>
<tr>
<td>Special requirements</td>
<td>none</td>
</tr>
<tr>
<td>Use case name</td>
<td>getMessages</td>
</tr>
<tr>
<td>---------------</td>
<td>-------------</td>
</tr>
<tr>
<td>Participating actor</td>
<td>none or 3rd Party Software</td>
</tr>
<tr>
<td>Entry condition</td>
<td>can be initiated only after the TraceProgram use case has been initiated.</td>
</tr>
<tr>
<td>Flow of events</td>
<td>Return the collection of messages, stored in the last run of the TraceProgram use case, for further usage.</td>
</tr>
<tr>
<td>Exit condition</td>
<td>none</td>
</tr>
<tr>
<td>Special requirements</td>
<td>none</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Use case name</th>
<th>Compute Invokations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Participating actor</td>
<td>none or 3rd Party Software</td>
</tr>
<tr>
<td>Entry condition</td>
<td>user should supply a collections of the classes, methods, fields and messages of the system, computed by Jrev, accessible by the above use cases.</td>
</tr>
<tr>
<td>Flow of events</td>
<td>1. check if all the needed collections of data is available. 2. compute the Invokation collection, as explained earlier.</td>
</tr>
<tr>
<td>Exit condition</td>
<td>none</td>
</tr>
<tr>
<td>Special requirements</td>
<td>none</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Use case name</th>
<th>get Invokations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Participating actor</td>
<td>none or 3rd Party Software</td>
</tr>
<tr>
<td>Entry condition</td>
<td>the “Compute invocations” use case should have been activated first.</td>
</tr>
<tr>
<td>Flow of events</td>
<td>1. return the collections of invocations which was previously computed by the “Compute Invokations” use case.</td>
</tr>
<tr>
<td>Exit condition</td>
<td>none</td>
</tr>
<tr>
<td>Special requirements</td>
<td>none</td>
</tr>
</tbody>
</table>
Use case name | Compute Dynamic Coupling
---|---
Participating actor | none or 3rd Party Software
Entry condition | user should supply a collections of the classes, methods, fields and messages of the system, computed by Jrev. Also the “Compute Invocations” use case should have been activated earlier.
Flow of events | 1. check if all the needed collections of data is available.
| 2. compute the Dynamic Coupling data, as explained earlier.
Exit condition | none
Special requirements | none

Use case name | get Coupling Results
---|---
Participating actor | none or 3rd Party Software
Entry condition | the “Compute Dynamic Coupling” use case should have been activated earlier.
Flow of events | 1. return the coupling data.
Exit condition | none
Special requirements | none

The static part of Jrev consists of only one actor, the User and three usecases, as can be seen in the Use Case diagram depicted in figure 4.3.
Below a description is given of each use case.

<table>
<thead>
<tr>
<th>Use case name</th>
<th>Init</th>
</tr>
</thead>
<tbody>
<tr>
<td>Participating actor</td>
<td>User</td>
</tr>
<tr>
<td>Entry condition</td>
<td>This use case uses the “Compute Static Coupling Measures”, “Compute Size Measures” and “Write Output” use cases. The system is not running yet, or is terminated from a previous execution.</td>
</tr>
<tr>
<td>Flow of events</td>
<td>1. The User writes the command he wants to execute on the command line with respect to the options explained previously. He presses “Enter” on the keyboard of the pc.</td>
</tr>
<tr>
<td></td>
<td>2. Check if user input is consistent and if a Java .jar file was supplied. Read in the collection of classes except those who are to be filtered</td>
</tr>
<tr>
<td></td>
<td>3. Incase the –cbo option is used, initiate the “Compute Static Coupling Measures” use case giving the previously collected classes.</td>
</tr>
<tr>
<td></td>
<td>4. If the –sizem option was used by the user, initiate the “Compute Size Measures” use case giving the collection of classes as parameter.</td>
</tr>
</tbody>
</table>
5. Write all derived measures in order to output by initiating the **Write Output** use case.


**Exit condition**
none

**Special requirements**
none

---

<table>
<thead>
<tr>
<th>Use case name</th>
<th>Compute Static Coupling Measures</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Participating actor</strong></td>
<td>none</td>
</tr>
<tr>
<td><strong>Entry condition</strong></td>
<td>This use case can only be initiated by the <strong>Init</strong> use case.</td>
</tr>
</tbody>
</table>
| **Flow of events**     | 1. Input to this use case is a collection of classes, of which thru the Bloat library the coupling measures CBO, NG_CBO, CBOImp, NG_CBO_Ump, CBOExp and NG_CBOExp are calculated.  
                          2. Return the results to the Init use case. |
| **Exit condition**     | none                             |
| **Special requirements** | none                           |

---

<table>
<thead>
<tr>
<th>Use case name</th>
<th>Compute Size Measures</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Participating actor</strong></td>
<td>none</td>
</tr>
<tr>
<td><strong>Entry condition</strong></td>
<td>This use case can only be initiated by the <strong>Init</strong> use case.</td>
</tr>
</tbody>
</table>
| **Flow of events**     | 1. Input to this use case is a collection of classes, of which thru the Bloat library the coupling measures NAImp and NMImp are calculated.  
                          2. Return the results to the Init use case. |
| **Exit condition**     | none                 |
| **Special requirements** | none              |
## Use case name

**Write Output**

## Participating actor

none

## Entry condition

This use case can only be initiated by the “Init” use case.

## Flow of events

1. Input to this use case is a collection of data which has to be written to an output file whose name is supplied.

## Exit condition

none

## Special requirements

none

---

### System decomposition

The system will be divided in the following subsystems:

1. **jrev**
2. **jrev.data**
3. **jrev.output**
4. **jrev.reverse.usecase**
5. **jrev.util.dcoupling**
6. **jrev.util.metrics**
7. **3rd party software**

The decomposition is made clear in figure 4.4.

**jrev.data**

This package contains all the core data structures used within the program. These include data structures for a class, method, object, message and collections of them. It is possible to use this structure in 3rd party software, and is present in the public API of Jrev.

**jrev.output**

This package contains functionality to understand the data structures in the **jrev.data** package, and to write them in a readable format to any output file. This package is used by any part of Jrev where data has to be written to an output file.

**jrev.reverse.usecase**

This package contains all he functionality to run the input-application to Jrev on the JVM and to trace its execution. It makes usage of the JDI library and the **jrev.data** package to store the retrieved data in some usable and specified structure for further usage in the rest of the program. This package contains a public API to retrieve the computed collections.

**jrev**

This package contains the main function of the program. It checks if the user input parameters are conform the specified usage of Jrev. Thereafter it uses the **jrev.reverse.usecase** package to run the input program and trace it. It retrieves the computed collections from that package, and
according to the user input parameters, uses the jrev.output package to output the collections to files or precede that with computing dynamic measures using the jrev.util.dcoupling package.

**jrev.util.dcoupling**
This package contains all the functionality to, given a collection of classes, methods, objects and messages, using the data structures of the jrev.data package, compute all the dynamic measures considered in this thesis.

**jrev.util.metrics**
This package contains a main function, which besides a path to a Java Jar file, takes some input parameters as input from which the to be computed static measures (the ones described in this thesis) are extracted and computed. The results are output to text files using the jrev.output package.

**3rd party software**
This package is a dummy package in the figure, to illustrate the possible dependencies of an external software application which uses Jrev’s functionality. In order to use Jrev’s results, it has to make usage of the jrev.data package. To trace another application, it has to make usage of the jrev.reverse.usecase package. If the dynamic measures are needed the jrev.util.dcoupling package is needed. The jrev.util.metrics package is used if static metrics are to be computed. Finally, if data structures of the jrev.data package are wished to be written to output, the jrev.output provides some basic functionality.
Object model – Class diagram

The internal decomposition of all the subsystems into classes and their dependencies to other packages are depicted in the several UML class diagrams beneath.

The jrev subsystem

Figure 4.5 is a class diagram depicting the jrev subsystem. The jrev package consists of the following classes:

- Jrev:
  This is the main class of Jrev. It checks user input parameters whether they are conform the specified usage. It uses the Filter class to filter user specified classes when collecting data. This class uses the `jrev.reverse.usecase` package to execute the input program. It then collects all the data (Classes, Methods, Objects and Messages), and thru the jrev.data package it writes this data to output files. If according to the user input dynamic measures were required, the `jrev.util.dcoupling` package will be used.

- Filter
  The Filter class is used to parse a String of class names, and add them to a collection of class names to be filtered by the system when executing and tracing the input program.

Figure 4.5: Class diagram of the Jrev package
The jrev.data subsystem

Figure 4.6 presents the UML class diagram of the jrev.data package. It contains the following classes:

- **myClass**
  Represents a java class. It contains the full name of the class (including package name).

- **myClasses**
  Contains a collection of myClass objects. When adding a java class to this collection, it checks whether this java class is already represented by a myClass object in the collection, else adds it to the collection.

- **myMethod**
  Represents a java method. It contains the name of the method, the class it is defined within, the formal parameters and the return type of the method.

- **myMethods**
  Contains a collection of myMethod objects. When adding a java method to this collection, it checks whether this java method is already represented by a myMethod object in the collection, else adds it to the collection.

- **myObject**
  Represents an instance of a Java object. It contains the unique identifier of the object and its class/type name.

- **myObjects**
  Contains a collection of myObject objects. When adding a java object to this collection, it checks whether this java object is already represented by a myObject object in the collection, else adds it to the collection.

- **myMessage**
  Represents a java message. It contains information regarding the from-object, the from-method, the to-object and the to-method.
The jrev.output subsystem

This subsystem contains only the Writer class and the SequenceThread class (figure 4.7). The Writer class provides basic functionality to output jrev.data data structures to files on the hard drive in a textual format. The SequenceThread class uses the software application “Sequence” which, given the sequence files from the jrev.util.dcoupling package (see the ‘-seq’ option) after tracing an application, shows a UML Sequence diagram graphically, which represents the data of those sequence files supplied.

Figure 4.6: UML Class diagram of the jrev.data package

Figure 4.7: UML Class diagram of the jrev.output package
The jrev.reverse.usecase subsystem

The class diagram of this package is given in figure 4.8. The jrev.reverse.usecase subsystem consists of the following classes:

TraceVM
The TraceVM class has the responsibility to initiate a JVM on which the input program will be running, with its parameters (supplied by the user). Having this user information, it initiates a TraceThread class for doing the trace using this initiated JVM. TraceVM further includes an API to access the data collections created by the underlying TraceThread.

TraceThread
This class extends the java.lang.Thread class in order to run in its own thread. It uses the JVM supplied by TraceVM, and uses the JDI library to inform the JVM which events to trace. For each thread the input application consists of, a unique EventHandler object is initiated to handle events private to this thread. Every event from the JVM is dispatched to the according EventHandler. When done executing the input application, all the information of all the EventHandlers is collected and made available the TraceVM object which initiated this TraceThread object.

EventHandler
The EventHandler class collects events send to it by the TraceThread, and handles these accordingly. It uses data structures from the jrev.data package to collect all the classes, methods, objects and messages of this thread of the input application.

TraceThread and EventHandler make usage of all the classes of the jrev.data package. And EventHandler make usage of the jrev.output.Writer class. But to keep the diagram simple these relations have been omitted.
Figure 4.8: UML Class diagram of the jrev.reverse.usecase
The jrev.util.dcoupling subsystem

Figure 24.9 and 4.10 depict a class diagram of the jrev.util.dcoupling subsystem. It consists of the following classes:

- **myInvocation**: This class represents an invocation. It contains data representing the from Method (and the from class) and the to method (and the to class).

- **myInvokations**
  This class is a collection (using the java HashMap) of myInvocation objects. It stores only distinct objects.

- **distinctMethod**
  This class represents a distinct method. It has a String attribute which is a concatenation of the from-class, the from-method, the to-class and the to-method.

- **distinctMethods**
  This class is a collection (using the java HashMap) of distinctMethod objects. It stores only distinct objects.

- **distinctClass**
  This class represents a distinct class. It has a String attribute which is a concatenation of the from-class, the from-method and the to-class.

- **distinctClasses**
  This class represents a collection (using a java HashMap) of distinctClass objects. It stores only distinct objects.

- **classLevel**
  This class represents a java class its coupling information. Therefore it contains the myClass object of the class it is representing, and distinctMethods ( IC_OM, IC_CM, EC_OM, EC_CM) and distinctClasses ( IC_OC, IC_CC, EC_OC, EC_CC) for storing the dynamic coupling of this myClass object. It also contains a string representation of its coupling information.

- **dcoupling**
  This class has the main algorithms for supplying the main functionality of this package. It takes as its constructor parameters a collection of myClass objects, a collection of myMethod objects, a collections of myObject objects and a collections of the dynamic messages of the system being measured. This data is supposed to be previously computed by Jrev. From this data, all invocations will be computed from the other data collections using the algorithm in chapter 2. It also provides functionality to compute coupling measures (IC_OM, IC_OC, IC_CM, IC_CC, EC_OM, EC_OC, EC_CM, EC_CC) for each myClass object (which are stored in classLevel objects).
Figure 4.9: UML Class diagram of the jrev.util.dcoupling part1
Figure 4.10: UML Class diagram of the jrev.util.dcoupling part2
**The jrev.util.metrics subsystem**

Figure 4.11 depicts a class diagram of the *jrev.util.metrics* subsystem. It consists of the following classes:

- **myClass**: This class represents a static Java Class. Using the Bloat library, the constructor takes an *EDU.purdue.cs.bloat.file.ClassFile* object as parameter, which is the Java class it represents. From this object, other information like the methods declared in the class, its fields and its super class are derived and made available thru its public interface.

- **myClasses**: The main functionality of this class is to gather a collection of distinct *myClass* objects by using a Vector object.

- **toProcess**
  This class its constructor takes as parameter a String object, which is a concatenation of Java class and/or package names, using the ‘;’ token as separator. The idea is that only these classes are to be used, others are excluded. If the input-string was empty, all classes are valid. To know whether a certain class is to be excluded or not, a function “validate” is made available.

- **Metrics**
  This class is the main class of this package, and acts as a stand alone application. It takes as a parameter a path to a Java JAR file. From this file, all classes it includes are extracted and added to a *myClasses* object if they satisfy the *toProcess* object this class uses for filter functionality. The functionality of the *myClass* class is used, to derive information like its methods and fields for each class added to the *myClasses* object. The next step is to make these classes available to other classes in this package to compute some static metrics (like *jrev.util.metrics.coupling.CBOcoupling*). Finally, the results are gathered and output to a text file. Size metrics like *NMImp* and *NAImp* are directly computed thru the information made available by each *myClass* object.

- **coupling.CBOcoupling**
  This class takes a *myClasses* object as parameter to its constructor, and using the Bloat library it computes the following static coupling and neighbor coupling measures: CBO, NG_CBO, CBOImp, NG_CBOImp, CBOExp and NG_CBOExp. The results are stored in a *cData* object, which is passed to the calling Metrics class, which writes these results to a file.

- **coupling.cData**
  This class is a data collection used by the *jrev.util.metrics.coupling*. The *CBOcoupling* class to store an Int array for all the metrics *CBOcoupling* computes. These values are handed to the Metrics class by using this class to hold all the results.
Figure 4.10: UML Class diagram of the jrev.util.metrics
Dynamic model – Sequence diagram

In Figure 4.11 the UML Sequence Diagram is depicted for the ExecuteProgram Use Case. Jrev is initiated by its main function. A Filter class is initiated, which parses the filter input String. Next, A TraceVM class is initiated in order to run and trace the input java application. This functionality is explained in the next Sequence Diagram. The results are retrieved from this TraceVM class and send to a decoupling object which uses this data to compute dynamic coupling measures. The Sequence diagram of this decoupling object is given below. The results are finally written to file or depicted thru a SequenceThread object which shows the generated sequence files in a graphical way.

In Figure 4.12 the UML diagram is given covering the functionality of the TraceVM object. In the beginning, a Java Virtual machine is started on which the input program is set to run. A TraceThread object is initiated to trace this execution. Therefore, the events to trace (thru JDI) are set and this Thread is started with the run call. At every run, the event queue is taken from the virtual machine, and handled accordingly. For every thread of the input program, a EventHandler object exists, which has to handle (log) specific events. In the end, the messages from these handlers are collected, and all the data is made available to the calling TraveVM object.

Finally figure 4.13 completes the functionality of the dynamic part of Jrev with the sequence diagram covering the functionality of the decoupling object. A decoupling object initiates a Vector of classLevel objects and a myInvocations object. The later is sued in the next step where according to the algorithm of Arisholm [3, 4 and 5] the invocations are computed using the messages of the input system. Next, for every traced class of the input java program, a classLevel object is initiated and its dynamic measures are computed. In the end, the collected metric data is made available to the caller.

Figure 4.14 shows the UML Sequence diagram regarding the static part of Jrev. The initiated Metric object uses a toProcess object to parse the input filter String for later usage. From the supplied Java jar file, the classes are extracted which pass the filter (validate). Every myClass object is stored in the myClasses object. If the “–order” options was used, the classes are ordered according to the supplied class file. A CBOCoupling object computes, given the collection of myClass object, all static coupling measures (CBO, NG_CBO, CBOImp, NG_CBOImp, CBOExp, NG_CBOExp). Size metrics are computed by the Metrics object it selves, and finally all the derived measures are output to file.
Figure 4.11: UML Sequence diagram of the ExecuteProgram use case.
Figure 4.12 UML Sequence Diagram of TraceVM. Continued on the next page.
Figure 4.13 UML Sequence Diagram of DCoupling
Figure 4.14 UML Sequence Diagram of the static part of Jrev
The public Jrev API

B.1 Package jrev.output

<table>
<thead>
<tr>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>public Writer( File file)</td>
</tr>
<tr>
<td>public Writer(String name)</td>
</tr>
<tr>
<td>public void close()</td>
</tr>
<tr>
<td>public void println(String string)</td>
</tr>
<tr>
<td>public void ShowSequence(File file)</td>
</tr>
<tr>
<td>public void writePNG (File file)</td>
</tr>
</tbody>
</table>

**public Writer( File file)**
constructor of the Writer class with a java File object as parameter, to which this Writer object will write data during its lifetime.

**public Writer(String name)**
constructor of the Writer class with String as parameter representing a path to a file, to which this Writer object will write data during its lifetime.

**public void close()**
flush the writer object, and afterwards close it safely.

**public void println(String string)**
lets the writer object write this parameter on a new line in the output file.

**public void ShowSequence (File file)**
Generate a visualization of the messages supplied by file in the form of a UML Sequence diagram. This function uses the functionality of the Sequence program (com.zanthan.sequence).

**public void writePNG (File file)**
Generate a PNG file visualalizing the messages supplied by file in the form of a UML Sequence diagram. This function uses the functionality of the Sequence program (com.zanthan.sequence).

B.2 Package jrev.data

<table>
<thead>
<tr>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>public myClass(ClassType classt)</td>
</tr>
<tr>
<td>public String toString()</td>
</tr>
</tbody>
</table>

**public myClass(ClassType classt)**

**public myClass(ClassType classt)**
constructor of the myClass class, taking a jdi ClassType object as parameter, from which additional information regarding this class can be extracted.

**public String toString()**
returns a string representation of this class.

<table>
<thead>
<tr>
<th>myClasses</th>
</tr>
</thead>
<tbody>
<tr>
<td>public myClasses()</td>
</tr>
<tr>
<td>public myClass getClass(ClassType ct)</td>
</tr>
<tr>
<td>public HashMap getMap()</td>
</tr>
</tbody>
</table>

**public myClasses()**
constructor of this class. Initiates a java HashMap in which all the classes will be stored.

**public myClass getClass(ClassType ct)**
Checks whether ct is already represented in the HashMap collection of myClass objects. If not, created a new myClass object with ct as parameter, and store it in the HashMap.

**public HashMap getMap()**
returns a HashMap object containing all the classes this myClasses object holds.

<table>
<thead>
<tr>
<th>myMethod</th>
</tr>
</thead>
<tbody>
<tr>
<td>public myMethod(Method m)</td>
</tr>
<tr>
<td>public String toString()</td>
</tr>
</tbody>
</table>

**public myMethod(Method m)**
constructor of this class, takes a jdi Method object as parameter from which other information will be retrieved.

**public String toString()**
returns a string representation of this method

<table>
<thead>
<tr>
<th>myMethods</th>
</tr>
</thead>
<tbody>
<tr>
<td>public myMethods()</td>
</tr>
<tr>
<td>public myMethod getMethod(Method m)</td>
</tr>
<tr>
<td>public Vector&lt;myMethod&gt; getMethods()</td>
</tr>
</tbody>
</table>

**public myMethods()**
constructor of this class. Initiates a HashMap for holding myMethod objects.

**public myMethod getMethod(Method m)**
Checks whether m is already represented in the HashMap collection of myMethod objects. If not, creates a new myMethod object with m as parameter, and stores it in the HashMap.
**public Vector<myMethod> getMethods()**
return a Vector of myMethod objects which were stored in the HashMap of this class.

### myObject

<table>
<thead>
<tr>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>public myObject(ObjectReference o)</td>
</tr>
<tr>
<td>public String toString()</td>
</tr>
</tbody>
</table>

**public myObject(ObjectReference o)**
constructor of this class. Takes a jdi ObjectReference object as parameter.

**public String toString()**
returns a string representation of this object.

### myObjects

<table>
<thead>
<tr>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>public myObjects()</td>
</tr>
<tr>
<td>public myObject getObjectId(ObjectReference or)</td>
</tr>
<tr>
<td>public Vector&lt;myObject&gt; getObjects()</td>
</tr>
</tbody>
</table>

**public myObjects()**
constructor of this class. Initiates a HashMap for holding myObject objects.

**public myObject getObjectId(ObjectReference or)**
Checks whether or is already represented in the HashMap collection of myObject objects. If not, create a new myObject object with or as parameter, and store it in the HashMap.

**public Vector<myObject> getObjects()**
returns a vector of myObject objects which were stored in the HashMap of this class.

### myMessage

<table>
<thead>
<tr>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>public myMessage(myObject fo, myObject to,myMethod fm, myMethod tm )</td>
</tr>
<tr>
<td>public String toString()</td>
</tr>
</tbody>
</table>

**public myMessage(myObject fo, myObject to,myMethod fm, myMethod tm )**
constructor of this class. Takes as parameter, the from-object of the message this class represents, the from-method, the to-object and the to-method.

**public String toString()**
returns a String representation of this message.
### B.3 Package jrev.reverse.usecase

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>public TraceVM(String mainName, StringBuffer args, String[] filter, int dbmode)</td>
<td>Initiates this class. Creates a JVM to execute the input program mainName with its parameters args in dbmode. The class names in filter will be filtered.</td>
</tr>
<tr>
<td>public Vector&lt;EventHandler&gt; getHandlers()</td>
<td>returns all the EventHandler objects created during execution of the input-program.</td>
</tr>
<tr>
<td>public Vector&lt;myMessage&gt; getMessages()</td>
<td>returns all the myMessage objects created during execution of the input-program.</td>
</tr>
<tr>
<td>public myObjects getObjects()</td>
<td>returns all the myObject objects created during execution of the input-program.</td>
</tr>
<tr>
<td>public myClasses getClasses()</td>
<td>returns all the myClass objects created during execution of the input-program.</td>
</tr>
<tr>
<td>public myMethods getMethods()</td>
<td>returns all the myMethod objects created during execution of the input-program.</td>
</tr>
</tbody>
</table>

### B.4 Package jrev.util.dcoupling

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>public myInvokation( String fromMethod, String toMethodorField)</td>
<td>creates a new myInvokation object. The parameters contains strings representing the fromMethod and toMethod/toField in the Method syntax explained earlier.</td>
</tr>
<tr>
<td>public String getToClass()</td>
<td>returns a string representation of the to class object.</td>
</tr>
<tr>
<td>public String getFromClass()</td>
<td>returns a string representation of the to class object.</td>
</tr>
</tbody>
</table>
public String toString()
returns a string representation of the this invocation. Eg : test2.c1.<init> --> test2.c4.<init>

<table>
<thead>
<tr>
<th>myInvocations</th>
</tr>
</thead>
<tbody>
<tr>
<td>public myInvocations()</td>
</tr>
<tr>
<td>public void putInvokation( String fromObject, String toObject, String fromMethod, String toMethodorField)</td>
</tr>
<tr>
<td>public HashMap getMap()</td>
</tr>
</tbody>
</table>

public myInvocations()
creates a new myInvocations object.

public void putInvokation( String fromObject, String toObject, String fromMethod, String toMethodorField)
Checks if there is not already a myInvokation object with the same parameters stored in this myInvokations object. If not, a new myInvokation object is created with the given parameters and stored.

public HashMap getMap()
returns the HashMap collections of all the myInvokation objects stored.

<table>
<thead>
<tr>
<th>distinctMethod</th>
</tr>
</thead>
<tbody>
<tr>
<td>public distinctMethod( String c1, String m1, String c2, String mf2)</td>
</tr>
<tr>
<td>public String toString()</td>
</tr>
</tbody>
</table>

public distinctMethod( String c1, String m1, String c2, String mf2)
creates a new distinctMethod object with the supplied parameters. c1 represents the from-class, m1 the from-method. c2 represents the to-class, and mf2 the destination method/field.

public String toString()
returns a string representation of this object.

<table>
<thead>
<tr>
<th>distinctMethods</th>
</tr>
</thead>
<tbody>
<tr>
<td>public distinctMethods()</td>
</tr>
<tr>
<td>public HashMap getMap()</td>
</tr>
<tr>
<td>public void addDistinctMethod( String c1, String m1, String c2, String mf2)</td>
</tr>
<tr>
<td>public String toString()</td>
</tr>
</tbody>
</table>

public distinctMethods()
creates a new object of this type.

public HashMap getMap()
returns the HashMap collections of all the distinctMethod objects stored.

public void addDistinctMethod( String c1, String m1, String c2, String mf2)
adds a distinctMethod object to the collection of there is not already one stored with the same parameters.
public String toString()
returns a string representation of this object.

distinctClass
public distinctClass(String c1, String m1, String c2)
public String toString()

public distinctClass(String c1, String m1, String c2)
creates a new object of this type. c1 represents the from-class, m1 the from-method and c2 the to-class.

public String toString()
returns a string representation of this object.

distinctClasses
public distinctClasses()
public HashMap getMap()
public void addDistinctClass(String c1, String m1, String c2)
public String toString()

public distinctClasses()
creates a new object of this type.

public HashMap getMap()
returns the HashMap collections of all the distinctClass objects stored.

public void addDistinctClass(String c1, String m1, String c2)
adds a distinctClass object to the collection of there is not already one stored with the same parameters.

public String toString()
returns a string representation of this object (all the stored distinctClasses).

classLevel
public classLevel()
public myClass getMyClass()
public String toString()
public distinctMethods getIC_OM()
public distinctClasses getIC_OC()
public distinctMethods getIC_CM()
public distinctClasses getIC_CC()
public distinctMethods getEC_OM()
public distinctClasses getEC_OC()
public distinctMethods getEC_CM()
public distinctClasses getEC_CC()
public class Level()
creates a new object of this type.

public MyClass getMyClass()
returns the MyClass object this object represents.

public String toString()
returns a string representation of this object.

public distinctMethods getIC_OM()
returns the collections of objects which anticipate in this particular measure.

public distinctClasses getIC_OC()
returns the collections of objects which anticipate in this particular measure.

public distinctMethods getIC_CM()
returns the collections of objects which anticipate in this particular measure.

public distinctClasses getIC_CC()
returns the collections of objects which anticipate in this particular measure.

public distinctMethods getEC_OM()
returns the collections of objects which anticipate in this particular measure.

public distinctClasses getEC_OC()
returns the collections of objects which anticipate in this particular measure.

public distinctMethods getEC_CM()
returns the collections of objects which anticipate in this particular measure.

public distinctClasses getEC_CC()
returns the collections of objects which anticipate in this particular measure.

dcoupling
public dcoupling(MyClasses classes, MyObjects objects, MyMethods methods, MyFields fields, Vector<MyMessage> messages)
public myInvokations getInvokations()
public Vector<classLevel> getClassLevels()
public void output(Writer w)
public void computeDynamicCoupling()
public int getIC_OM()
public int getIC_OC()
public int getIC_CM()
public int getIC_CC()
public int getEC_OM()
public int getEC_OC()
public int getEC_CM()
public int getEC_CC()
public decoupling(myClasses classes, myObjects objects, myMethods methods, myFields fields, Vector<myMessage> messages)
creates a new decoupling object with as parameters the collection of classes, objects, methods, fields and messages. During initialization, a new collections is generated using the supplied data, representing all the invocations of the input system

public myInvokations getInvokations()
returns the collection of computed myInvokations objects.

public Vector<classLevel> getClassLevels()
returns the collection of classLevel objects. This contains one classLevel object for each myClass object supplied during initialization. Use the computeDynamicCoupling call first.

public void output(Writer w)
writes to w all dynamic coupling data collected including total coupling, per class, and the collection of invocations.

public void computeDynamicCoupling()
computes for each myClass object supplied during initialization, a according classLevel object, and computes its dynamic coupling measures.

public int getIC_OM()
returns the total IC_OM coupling measured of this system. Call computeDynamicCoupling first in order to compute all dynamic coupling data.

public int getIC_OC()
returns the total IC_OC coupling measured of this system. Call computeDynamicCoupling first in order to compute all dynamic coupling data.

public int getIC_CM()
returns the total IC_CM coupling measured of this system. Call computeDynamicCoupling first in order to compute all dynamic coupling data.

public int getIC_CC()
returns the total IC_CC coupling measured of this system. Call computeDynamicCoupling first in order to compute all dynamic coupling data.

public int getEC_OM()
returns the total EC_OM coupling measured of this system. Call computeDynamicCoupling first in order to compute all dynamic coupling data.

public int getEC_OC()
returns the total EC_OC coupling measured of this system. Call computeDynamicCoupling first in order to compute all dynamic coupling data.

public int getEC_CM()
returns the total EC_CM coupling measured of this system. Call computeDynamicCoupling first in order to compute all dynamic coupling data.

public int getEC_CC()
returns the total EC_CC coupling measured of this system. Call `computeDynamicCoupling` first in order to compute all dynamic coupling data.