Investigation of the impact of cohesion on the change-proneness of Java interfaces

Master's thesis

René Pingen

Investigation of the impact of cohesion on the change-proneness of Java interfaces

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

René Pingen
born in Haarlemmermeer, the Netherlands

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

Investigation of the impact of cohesion on the change-proneness of Java interfaces

Author: René Pingen
Student id: 1263080
Email: R.A.Pingen@student.tudelft.nl

Abstract

A lack of cohesion is often associated with bad software quality, and could lead to more changes and bugs in software. In this thesis the impact of cohesion on the change-proneness of Java interfaces is investigated. Showing the existence of a relation between these concepts can lead to better change prediction models that can support software developers in defect prediction and prevention tasks. An empirical study is performed on several open source projects to test three hypotheses.

The first hypothesis investigates whether cohesion metrics correlate with the number of fine-grained source code changes. The results of the correlation analysis show a correlation between two cohesion metrics and the number of changes in Java interfaces.

The confounding effect of class size is a possible explanation for the correlation between the cohesion metrics and the number of fine-grained changes. This idea is investigated through the second hypothesis, which studies the correlation between the cohesion metrics and interface size metrics. The hypothesis is accepted for the same two metrics.

The third hypothesis of this thesis tries to answer whether cohesion metrics can improve change prediction models based on size. By performing three different experiments with multiple classification algorithms, we have found no evidence that supports the final hypothesis.

Concluding, cohesion metrics can be used to predict changes in source code. However, they are not better predictors than size metrics, and we have found no evidence to support the idea that they can improve change prediction models based on size.

Thesis Committee:

Chair:	Prof.dr.ir. G.J.P.M. Houben, Faculty EEMCS, TU Delft
University supervisor:	Dr. M. Pinzger , Faculty EEMCS, TU Delft
Committee Member:	Dr.ir. A.J.H. Hidders, Faculty EEMCS, TU Delft

Preface

This report is the result of my Master's thesis project, which I performed at the Software Engineering Research Group (SERG). In this thesis project, I got the opportunity to perform research in the field of Software Engineering. Based on three courses I followed in the Master of Science curriculum, I was able to quickly dive into the research field and start my thesis project. I would like to thank several people that helped making this project possible.

After following the software architecture course, I became very interested in software architecture and software quality. In an initial meeting with Martin Pinzger, we discussed many different possibilities for a graduation project. As I became enthusiastic about the research field, I followed the software re-enigineering and seminar courses presented by Martin Pinzger and Andy Zaidman. After a few more meetings with Martin, I was able to start my thesis project at the end of April 2011.

First of all, I would like to thank Martin Pinzger for his guidance and assistance during this thesis. In nearly every meeting, we discussed new research ideas that motivated me to investigate them immediately. This motivation made sure that we performed many different experiments, leading to the thesis research presented in this report. Furthermore, I would like to thank Daniele Romano for his assistance and feedback during my research.

I would like to thank Geert-Jan Houben and Jan Hidders for giving me the opportunity to perform my thesis research at SERG. On several occasions, Jan Hidders asked me critical questions about my research. As a member of a different research group, Jan Hidders provided valuable feedback from a different point of view.

During my research, I was able to discuss my research with many other students with different backgrounds, and some of them reviewed draft versions of this report. I would like to thank all the people who helped me with this thesis.

René Pingen
Delft, the Netherlands
March 14, 2012

Contents

List of Figures

Part I

Introduction

Chapter 1

Introduction

1.1 Introduction

When a software product is released, it often still contains bugs and problems that have to be fixed. It has been estimated that software maintenance costs account for forty to eighty percent of the total software development costs [19]. This implicates that reducing the need for maintenance could prove to be very profitable.

We think that improving software quality can reduce maintenance costs. This leads to the following questions: what is software quality, and how can we improve it? There are numerous ideas and theories about good software design and quality, but do these ideas lead to better software?

One of these ideas is that software should be *"Loosely coupled and highly cohesive"*. Stevens et al. defined the concepts of coupling and cohesion in 1979:

- Coupling can be defined as *the measure of strength established by a connection from one module to another* [40].
- Cohesion can be defined as *a measure of the degree to which the elements of a module belong together* [40].

The goal of this research is to investigate the impact of cohesion on the change-proneness of Java interfaces. Our expectation is that non-cohesive classes are more likely to be change-prone than cohesive classes. Note that this research focuses on the cohesion in interfaces, which includes Java interface classes and the interfaces of concrete classes (i.e. the exposed methods of concrete classes).

Figure 1.1 illustrates the global idea of this research. We study a hypothesized relation between cohesion and software quality by investigating the relation between cohesion metrics and the number of changes in a class. The cohesion metrics that are investigated in this research are described in Chapter 3.

We perform an empirical study using several open source projects to investigate the relation between cohesion and the change-proneness of interfaces.

The next section describes the motivation of this research. Section 1.3 formulates the research questions, and Section 1.4 describes the structure of this report.

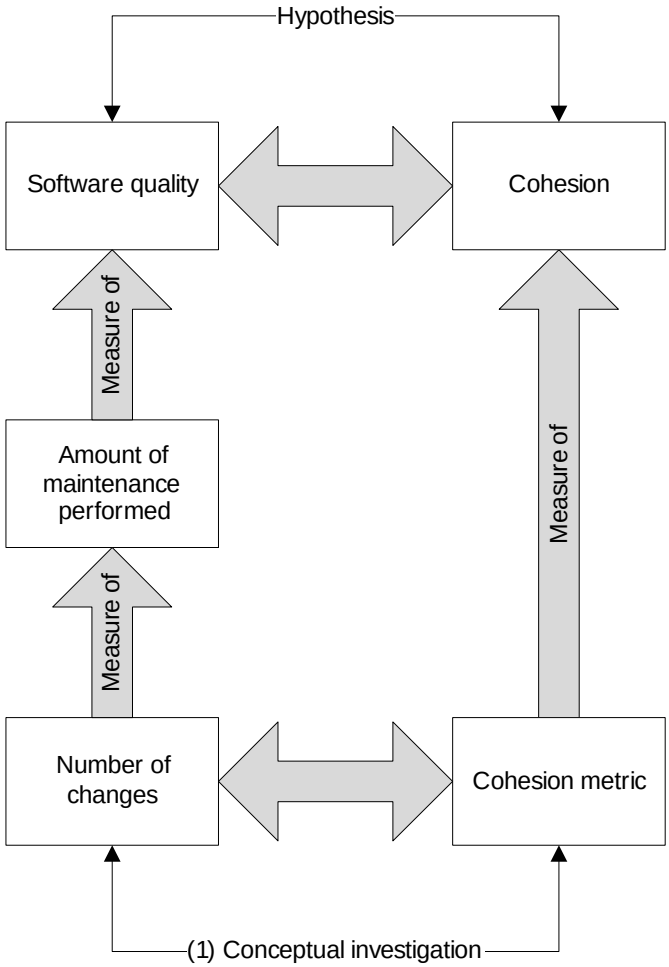


Figure 1.1: Research idea

1.2 Research motivation

As described in the previous section, the goal of this research is to investigate the impact of cohesion on the change-proneness of Java interfaces.

An interface can be seen as a contract between two components. If this contract is changed for some reason, every class that uses or implements the interface has to adapt to that change. In effect, this means that one change can lead to numerous other changes in other parts of the system. Therefore, we think that reducing the amount of interface changes can lead to more stable software.

There are many reasons for an interface to change. A lack of cohesion is one of the possible reasons we investigate in this thesis. If we find a relation between cohesion and the change-proneness of interfaces, we can use this information to predict and maybe

even prevent future changes.

Currently, many change prediction models are based on size metrics [18, 45]. These models are based on the idea that larger classes change more frequently than smaller classes. Better change prediction models can lead to better change- and fault-detection tools. This can eventually lead to a reduction in software maintenance and development costs.

Previous work by Romano and Pinzger [36] shows a relation between the Interface Usage Cohesion (IUC) metric and the change-proneness of Java interfaces. We extend the work by Romano and Pinzger by including more cohesion metrics, investigating more projects, and performing more extensive experiments.

1.3 Research questions

The relation between cohesion metrics and the change-proneness of classes is investigated with an empirical study. We apply several statistical methods on the data of the selected projects to test our hypotheses. The first hypothesis of this research is:

> H_1: The cohesion metrics are correlated with the number of fine-grained changes in Java classes.

Besides interface classes, we look at the interfaces of concrete classes as well. We have two reasons for including them: As we expect different usage and change behavior of interfaces and concrete classes, it is interesting to analyze the differences between interfaces and concrete classes. The second reason for analyzing concrete classes is that it allows us to work with much larger datasets.

Previous research has shown that the number of changes in a class is related to the size of the class[13, 18, 33, 44, 45]. If we assume a uniform distribution of changes in the source code of a project, then indeed larger classes will have more changes.

If there is a correlation between the number of changes and a cohesion metric, that correlation could be explained by the fact that there is another correlation between the cohesion metric and the size of a class. In other words, do less cohesive classes change more often because they are less cohesive, or just because they tend to be larger? Obviously we cannot easily prove or disprove this causal relation, but we can investigate how the different metrics relate to each other. This leads to the definition of the second hypothesis:

> H_2: The cohesion metrics are correlated with the size of Java classes.

The correlation coefficients indicate just one aspect of the relation between the metrics and the changes. An interesting question is whether cohesion metrics can improve change prediction models:

> H_3: The cohesion metrics can improve the performance of prediction models to classify Java classes into change- and not change-prone.

To evaluate this final hypothesis we will employ a number of machine learning techniques.

1.4 Structure

This report is split into four parts. The overview of the research structure is shown in Figure 1.2.

The first part starts with the introduction, which describes the goal, research questions, motivation and structure of the thesis. The second chapter describes related work in the research field and provides background information. Chapter 3 describes the selection and definition of the cohesion metrics investigated in this thesis.

The second part describes how the research questions are investigated. Chapter 4 describes the research framework, and how the data required for the investigations is acquired. Chapter 5 describes the approach to the first two hypotheses. It describes how the correlation analyses are performed, and how the results should be interpreted. Similarly, Chapter 6 describes how the third hypothesis is investigated using classification models.

Part III describes the execution of the empirical study. In this part, Chapter 7 describes the selection of projects for the empirical study. Then, Chapters 8 and 9 describe the results of respectively the correlation analysis and the prediction models.

Finally, Part IV concludes this thesis. Chapter 10 discusses the results of the empirical study and the threats to the validity of these results. Conclusions and future work are described in the final chapter.

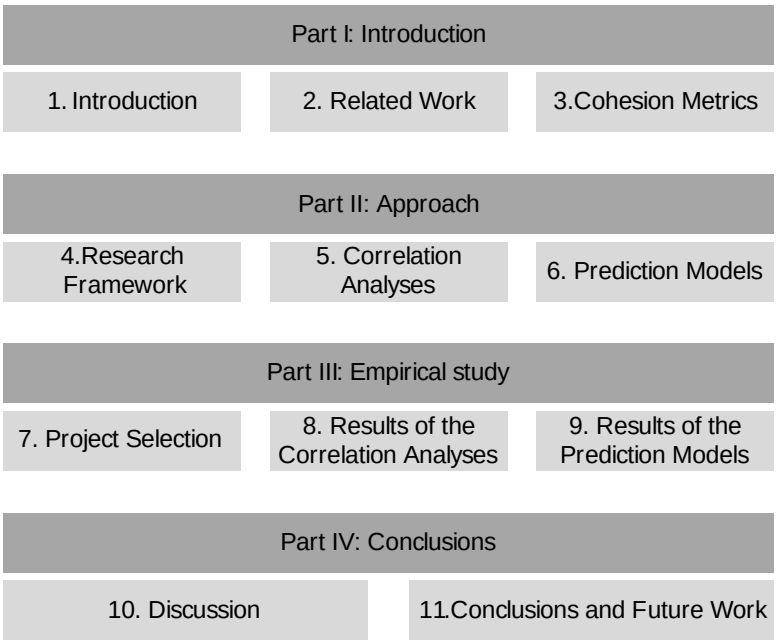


Figure 1.2: Thesis structure

Chapter 2

Related Work

Much research has been done on source code metrics and their relation with the change- and fault-proneness of source code. In this chapter, we give an overview of the research that has been done in this field, and the contributions of this research.

2.1 Overview

As an introduction to some of the concepts and terminology used in this thesis, Section 2.2 provides background information. Section 2.3 describes research that investigates software design and the impact on change- and fault-proneness. In Section 2.4, related work on the impact of software design on quality aspects other than change- and fault-proneness.

The final section discusses the contributions of this research, in relation to the related work. Research on cohesion metrics is described in the next chapter.

2.2 Background

2.2.1 Modularization, Coupling and Cohesion

Modularization is an important concept in object-oriented programming. Parnas [34] discussed criteria that can be used to decompose systems into modules. In his paper, he coined the term of *information hiding*. Information hiding is the idea of hiding different design decisions from another. If a design decision is changed, this means that only one module has to change instead of multiple modules.

Stevens et al. [40] argue that strong coupling between modules complicates a system. By reducing coupling between modules, the number of paths along which changes and errors can propagate are reduced as well. Coupling is defined as *the measure of strength established by a connection from one module to another* [40].

Similarly, cohesion is defined as *a measure of the degree to which the elements of a module belong together* [40]. It is argued by Eder et al. [12] that cohesion is one of the most important software quality criteria. "Modules with strong cohesion, are easier to maintain, and furthermore, they greatly improve the possibility for reuse" [12]. Stevens

et al. [40] identified six categories of module cohesion, later extended by Yourdon et al. [43]: Coincidental, Logical, Temporal, Procedural, Communicational, Sequential and Functional. They give an indication of the cohesiveness starting from coincidental (weak) to functional (strong).

Eder et al. [12] identify three types of cohesion: Method cohesion describes how good the elements of a method belong together, whereas class cohesion describes how well the elements of a class belong together. Finally, inheritance cohesion takes into account the inheritance hierarchies.

2.2.2 Object-Oriented design patterns and principles

In software design, a design pattern is a re-usable solution that can be applied in commonly occurring situations. The idea of design patterns originated in architecture, where it was introduced by Christopher Alexander in 1977 [2]. In software engineering, the book by Gamma et al. [17] introduced the concept of design patterns in software engineering to a wider public. In practice, a design pattern is a known good solution for specific situations. If a designer recognizes a situation and the appropriate pattern, applying the pattern will be a good an quick solution. If he applies a wrong pattern for the wrong situation, this will obviously not be the case.

In contrast to design patterns, design principles do not provide concrete solutions. Instead, design principles are guidelines that should be followed in software design. In [27], Martin describes several of these principles. One of these design principles is the Interface Segregation Principle (ISP), which states that: "Clients should not be forced to depend upon interfaces they do not use". More concrete, if a client of an interface only invokes half of the methods of the interface, it might be an idea to split the interface. The Interface Usage Cohesion (IUC) metric we investigate in this thesis is related to this principle.

Another design principle proposed by Martin is the Single Responsibility Principle (SRP). It states that: "There should never be more than one reason for a class to change" [27]. Martin argues that if a class has multiple responsibilities, it will have more reasons to change. Note that SRP is related to cohesion, as classes with multiple responsibilities are by definition non-cohesive.

2.3 Software design and the impact on change- and fault-proneness

This thesis research is based on earlier work by Romano and Pinzger [36]. In their work, they show a relation between the IUC metric and the change-proneness of Java interfaces. The relation between software design and change- and fault-proneness has been researched intensively. In 1993, Li and Henry investigated the relation between several object-oriented metrics and required maintenance effort in terms of Lines Modified (LM) [26]. They found that there is a relation between the two, and that the required maintenance effort can be predicted using source code metrics. Similar results are found

by Khoshgoftaar and Szabo [25] in 1994, who use several metrics to train a neural network that can predict the required maintenance.

More recently, Khomh et al. investigate the relation between code smells and software change-proneness [23]. Their study shows that classes with code smells are more change-prone than classes without code smells. Similarly, Khomh et al. investigate the impact of anti-patterns on class change- and fault-proneness [24]. In this study they show that anti-patterns have an impact on class change- and fault-proneness. Interestingly, they also find that size alone cannot explain the higher odds of classes with anti-patterns of being changed.

The relation between code metrics and size is investigated further in several papers. El Emam et al. [13] find a relation between several object-oriented metrics and the fault-proneness of classes. However, when controlled for class size, this relation disappears. Olbrich et al. [33] study god and brain classes in the evolution of three open source systems. Their study shows that god and brain classes do tend to change more often and contain more defects than other classes. However, when normalized for size, the classes show fewer changes and fewer defects. Similar results are found by Zazworka et al. [44].

Similarly, Zhou et al. investigate the potentially confounding effect of class size on the associations between object-oriented metrics and change-proneness [45]. They show that for many metrics, the relation between the metric and class size can completely explain their relation with the change-proneness of classes. Interestingly, the set of metrics investigated by Zhou includes cohesion metrics we include in this research as well. In Chapter 10, we will compare the results of Zhou [45] with our own results.

Moser et al. [29] compare the performances of change metrics and source code metrics as fault-predictors. Their results suggest that change metrics are more effective change predictors than complexity metrics. In other words, a class that has changed in the past is likely to change in the future, and, a class that is complex does not mean it will change often.

The investigation of D'Ambros et al. [9] shows a relation between the presence of design flaws and software defects. They find that that an increase in design flaws is related to an increase in bugs, although they could not point out one design flaw that consistently correlates more than others with the number of bugs.

2.3.1 Defect prediction

Im some cases, changes in software can cause new bugs. Sliwerski et al. [39] describe how to automatically detect these changes, and find that fix-inducing changes show distinct patterns. They find that fix-inducing changes can be up to three times the size of a non-fix-inducing change.

Hassan [20] tests the idea that if a file has more complex changes, the chances are higher it will contain faults. Through an empirical study of six open source projects, he finds that the number of prior faults is a better predictor of future faults in comparison to the number of prior modifications.

Nagappan et al. [30] find in an empirical study that failure-prone software entities are statistically correlated with code complexity measures. However, they could not find a universal set of defect predicting complexity metrics. Furthermore, they show that a set of metrics that can effectively predict failure-prone entities is project dependent. Similarly, [47] investigates the idea of cross-project defect prediction. The results show that it is very difficult to apply prediction models on another project than they were constructed for.

Nagappan and Ball [31] show that system defect density can be predicted using relative code churn measures. Code churn measures the extent of change made to a component over a period of time.

Furthermore, Nagappan et al. [32] introduce the concepts of change- bursts. A change-burst occurs when the same code changes often in a period of time. Naggapan shows that change-bursts have a high predictive power for defect-prone components.

Zimmerman et al. [46] have researched the detection of change-couplings. If two entities often change in the same transaction, they are coupled through their changes. This information can be used to guide programmers, and make suggestions to the programmer (e.g. if you change class A, there is a high probability you should change class B as well). Zimmermann presents the ROSE tool which is able to make these kinds of suggestions.

In [38], Schröter et al. construct prediction models with failure history and design data to predict post-release failures. They find that the likelihood of a component to fail is significantly determined by the set of components that it uses.

2.4 Software design and the impact on other software quality aspects

The impact of software design on and software quality aspects other than the change- and fault-proneness have been researched as well. For example, Rombach [37] investigated the impact of software structure on comprehensibility, locality, modifiability and reusability. Deligiannis et al. study the effect of god classes on the maintainability of software systems [10]. The experiment is performed by several students and questionnaires seem to support the hypothesis that god classes are harder to understand. Similarly, Du Bois et al. investigated the effect of god classes on program comprehensibility [11]. The results of their study show evidence that comprehensibility is affected by the presence of god classes, although the precise effects are not clear.

The research performed by Abbes et al. [1] investigates the relation between the presence of anti-patterns and program comprehensibility. The results show that programmers usually can cope with a single anti-pattern, but that comprehensibility is affected by a combination of two anti-patterns.

2.5 Contributions of this work

Similar to most work described in this chapter, our research has the goal to assess whether source code metrics have an impact on the change-proneness of software. More generally, we try to validate our ideas about software quality in general. This work differentiates itself in at least two ways. First of all, this research is focused on the investigation of the cohesiveness of interface classes. Many of the work described in this chapter focuses on the investigation of code smells and anti-patterns, where we focus specifically on cohesion metrics for interfaces. Note that Zhou et al. [45] include several cohesion metrics, which we investigate as well. Based on the research by Romano and Pinzger [36], we include the IUC metric.

Furthermore, in this research we investigate fine-grained Source Code Changes (SCC). Much of the discussed work investigates the number of lines modified (also called code churn), or the number of commits in a repository. Using the Evolizer framework (see Chapter 4), we are able to calculate the changes between each revision, and distinguish between significant and non-significant changes.

Chapter 3

Cohesion Metrics

In the previous chapter, the concepts of coupling and cohesion are described. Recall the definition of cohesion used in this thesis: *Cohesion is a measure of the degree to which the elements of a module belong together* [40].

This chapter describes the definition of cohesion metrics for interfaces. These metrics allow us to give an indication of the cohesiveness of an interface. Note that we focus on interface cohesion, which means we pay attention only to the externally exposed method signatures within Java interfaces and classes, while ignoring method bodies and attributes.

3.1 Overview

The main focus of this thesis is on the IUC metric, which is defined in Section 3.2. The choice for this metric is based on the promising results of the previous study by Romano and Pinzger [36]. Based on the research of Perepletchikov et al. [35] on service interface cohesion, we analyze three different types of cohesion:

3.2. Usage cohesion

3.3. Data cohesion

3.4. Implementation cohesion

The last section of this chapter presents an overview of the selected cohesion metrics.

3.2 Usage cohesion

As the name indicates, usage cohesion metrics measure the cohesiveness of interfaces, based on how they are used. A class C_1 is considered to be a client of class C_2 when some method in C_1 invokes at least one method on an instance of C_2. Similarly, the usage of a class by a client can be defined as how many of the class methods the client invokes.

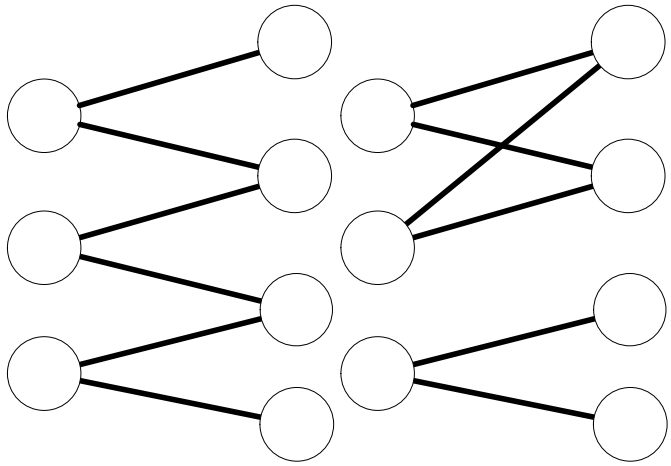
3.2.1 Interface Usage Cohesion

The Interface Usage Cohesion (IUC) metric [35, 36] is a measure of how much clients use the interface. For each client, the number of methods invoked divided by the total Number of Methods (NOM) indicates the interface usage ratio. If we divide the sum of these ratios by the Number of Clients (NOC), IUC is calculated.

$$IUC = \frac{\sum_{i=1}^{NOC} \frac{\text{used methods}}{NOM}}{NOC}$$

If all clients invoke all of the methods of a class, $IUC = 1$. If a class has no clients, IUC is undefined. This decision means $min(IUC) = \frac{1}{NOM}$, making the range of IUC $[\frac{1}{NOM} - 1]$.

The IUC metric has several limitations. First of all the IUC metric is an average of the usage ratios. Averages give no information about the distribution of the data. A second limitation is the fact that IUC does not reveal anything about the similarity between the clients. This is illustrated in Figure 3.1 where in both situations IUC=0.5. While it can be discussed if one of the classes is more cohesive than the other, it is an aspect that could or maybe even should be included in a cohesion metric.



1: IUC=0.5 LCOIC=0 2: IUC=0.5 LCOIC=1

Figure 3.1: Same IUC, different cohesion?

3.2.2 Lack of Cohesion of Interface Clients

An alternative to IUC is the Lack of Cohesion of Interface Clients (LCOIC) metric. It is based on the original Lack of Cohesion of Methods (LCOM) metric defined by Chidamber & Kemerer [6] which is described in Section 3.3.2.

The difference with the original LCOM is that we use LCOIC to measure the cohesion of an interface through the method usage of its clients, instead of measuring the cohesion of a class through the attribute usage of its methods. LCOIC is defined as follows:

```
P=0;Q=0;
for each pair of clients
{
  if shareMethods(Client1,Client2)
   Q++;
  else
    P++;
}
LCOIC=(P>Q) ? P-Q : 0;
```

The idea is that if the sets of invoked methods two clients share any method, Q is increased. Else, P is increased. Whenever P is larger than Q, the class is considered not cohesive, and the value of LCOIC is P-Q. In other cases, LCOIC is 0, and thus the range of LCOIC is $[0 - \infty]$. In the situation sketched in Figure 3.1, LCOIC=0 in Figure 3.1.1, and LCOIC=1 in Figure 3.1.2.

Several limitations of LCOM have been identified which apply to LCOIC as well. First of all, the metric assigns a value of zero to very different classes. Similarly, the metric can assign the same value to very different violations. To overcome these limitations, several alternative metrics have been constructed [5, 21, 22]. These alternatives include a metric that is (if applied to interface usage) similar to IUC.

3.3 Data cohesion

Another aspect that can be used to measure the cohesiveness of classes is data cohesion. Methods in classes access data in several ways, such as through their parameters or through field accesses. If all methods of a class operate on the same data, a class can be considered cohesive in terms of data usage.

3.3.1 Parameter usage cohesion

There are several different metrics that can be constructed to measure the cohesiveness of an interface through the parameter usage of its methods. An example of this is Cohesion among Methods in a Class (CAMC), proposed by Bansiya et al [4]. It is calculated by simply counting all distinct parameter types in a class, and dividing that by the product of all methods and parameters. If all methods use all data types $CAMC = 1$, in other cases CAMC will have any value between $[0 - 1]$.

$$CAMC = \frac{\sum_{i=1}^{NOM} \text{used params}}{NOM \cdot NOP}$$

In the original definition, CAMC is the average of the entries in the parameter occurrence matrix. When the formula is rewritten, it becomes visible that CAMC is

very similar to IUC. However, there are a few decisions that influence the calculation of CAMC:

- We have chosen the CAMC metric that does not include the datatype of the container class in the parameter list.
- Primitive types are included in the calculation.

CAMC suffers from limitations similar to the limitations of IUC, as described by Counsell et al. [7]. Furthermore, there are several other limitations that are worth mentioning. First of all, getters and setters are not recognized as similar, as the return type is not included in the parameter list. A solution could be to look at the function names and/or return types. Second, only looking at the data types assumes that every use of a data type is the same. Two methods that both have a string as input, could be completely unrelated. To resolve this, it might be worth to look at parameter names. Including parameter names can also solve the third limitation: multiple occurrences of data types are ignored.

Counsell et al. [7] propose another metric similar to CAMC. The Normalized Hamming Distance (NHD) metric calculates the average hamming distance over the parameter occurrence matrix. In other words, it calculates the *parameter agreement* between each pair of methods. Formally:

$$NHD = \frac{2}{lk(k-1)} \sum_{j=1}^{k-1} \sum_{i=j+1}^{k} a_{ij}$$

Each of the l columns of the parameter occurrence matrix represents a unique data type, and the k rows represent the methods. The value of a_{ij} is the number of entries in rows i and j that are equal. The values of NHD are between $[0-1]$. Counsell et al. [7] show that NHD can be formulated differently, which can be easier to compute:

$$NHD = 1 - \frac{2}{lk(k-1)} \sum_{j=1}^{l} c_j(k-c_j)$$

where c_j is the number of 1s in the jth column of the parameter occurrence matrix.

NHD has limitations similar to CAMC. First of all, it gives similar values for classes that seem to be different in terms of cohesiveness [8]. Furthermore, the limitations for primitive types, getters and setters and return types apply to NHD as well.

3.3.2 Attribute usage cohesion

The attribute usage of methods can be used as a cohesion metric for classes that implement methods. Chidamber & Kemerer [6] defined Lack of Cohesion of Methods (LCOM), which is similar to the definition of LCOIC as discussed in Section 3.2.2. The difference is that the methods now are the clients, and for each pair of methods we check whether they share attributes or not. In pseudo code:

```
P=0;Q=0;
for each pair of methods
{
  if shareAttributes(Method1,Method2)
   Q++;
  else
    P++;
}
LCOIC=(P>Q) ? P-Q : 0;
```

The limitations of LCOM are described in several publications [5, 21, 22], and several variations and alternatives for this metric have been proposed. As the focus of this paper lies on the cohesiveness of interfaces, we will not describe them here.

3.4 Implementation cohesion

Implementation cohesion metrics measure the cohesiveness of a class based on how interfaces are implemented. Similar to the SSIC metric defined by Perepletchikov et al. [35], we define the Interface Implementation Cohesion (IIC) metric. The IIC metric measures how much of an interface is actually implemented by classes. Java forces concrete classes to implement all abstract methods that are declared in its inheritance tree, but not yet implemented. Although this makes sure all interfaces are completely implemented, in practice empty implementations often occur.

IIC is calculated by dividing the number of methods by the Number of Implementing Classes (NOIC) for each implementer, and dividing the sum by the number of implementing classes. If an interface is completely implemented, IIC=1.

$$IIC = \frac{\sum_{i=1}^{NOIC} \frac{\text{implemented methods}}{NOM}}{NOIC}$$

Note that this definition of IIC is similar to IUC and CAMC.

There are several aspects to consider when calculating IIC:

- Only concrete classes that appear in the inheritance tree of the interface are included as implementing classes.
- To find which class in the inheritance tree implements a function, the inheritance tree is walked upwards starting at the concrete class.
- For interfaces that do not have implementing classes IIC is undefined.

A consequence of this approach is that when an abstract class completely implements an interface, it can be included multiple times in the metric if it is extended by more than one concrete class.

3.5 Overview of cohesion metrics

In the previous sections several cohesion metrics have been described. Table 3.1 gives a short overview of the metrics. The next chapters will investigate the relation between these metrics and the number of changes in interfaces and classes.

Type	Metric	Description
Usage cohesion	IUC	Interface Usage Cohesion
	LCOIC	Lack of Cohesion of Interface Clients
Data cohesion	CAMC	Cohesion Among Methods in a Class
	NHD	Normalized Hamming Distance
	LCOM	Lack of Cohesion of Methods
Implementation Cohesion	IIC	Interface Implementation Cohesion

Table 3.1: Overview of cohesion metrics.

Part II

Approach

Chapter 4

Research Framework

The goal of this thesis is to investigate the relation between cohesion metrics and the change-proneness of Java interfaces and classes. The investigation is formalized through the definition of hypotheses in Section 1.3. This chapter describes the research framework used to investigate the hypotheses.

4.1 Overview

The research framework of the empirical study can be broken down in three components:

1. Source code metric computation (Section 4.2)
2. Source Code Changes (SCC) extraction (Section 4.3)
3. Correlation and prediction analysis (Chapter 5 and Chapter 6)

Figure 4.1 is an illustration of the research framework. Most of the tasks described in this chapter are performed using the Evolizer framework [16].[1] The Evolizer framework provides functionalities to extract a meta-model from Java source code, as is described in Section 4.2.1. This model can then be used to calculate the source code metrics.

To compute the fine-grained changes, Evolizer can import file revisions from software versioning systems (Section 4.3.1). This information is then used by the Change Distiller to extract fine-grained source code changes (Section 4.3.2).

Before the experiments to investigate the research questions can be performed, the metrics have to be mapped to the changes, which is described in Section 4.4. The correlation analyses that are used to answer the first two hypotheses are described in Chapter 5, the prediction models are described in Chapter 6.

[1] The Evolizer framework, http://www.evolizer.org

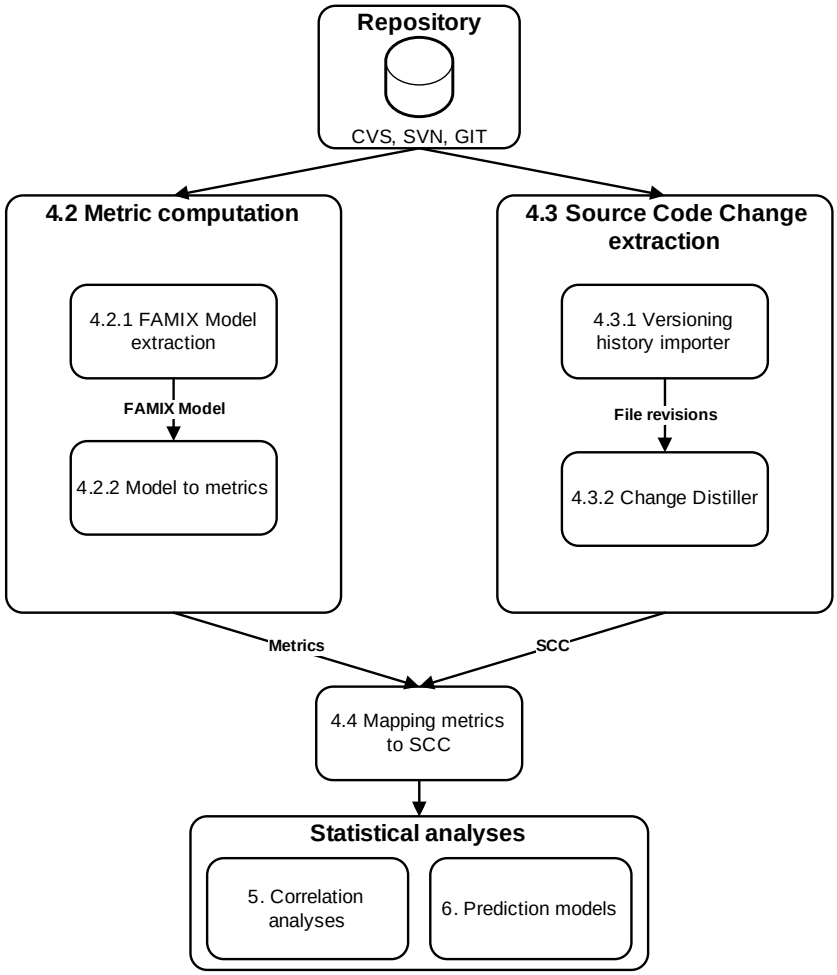


Figure 4.1: An overview of the research framework

4.2 Metric computation

The computation of source code metrics consists of a three step process. First, the source code of the selected project has to be imported in Eclipse.[2] Then, using the Evolizer FAMIX Importer, a FAMIX meta-model can be extracted from the source code. The FAMIX model is a representation of the source code which allows us to easily compute the source code metrics. Finally, the model can be used to calculate the source code metrics.

[2]The Eclipse project, http://www.eclipse.org

4.2.1 FAMIX model extraction

The Evolizer framework contains functionalities to extract a FAMIX meta-model from a Java project. Using Java source code as input, a FAMIX model can be extracted. A FAMIX model can be seen as a description of the static structure of a projects source code. Figure 4.2 shows the simplified structure of the FAMIX model, as it is implemented in the Evolizer framework. A complete UML diagram of the FAMIX model can be found at the moose website.[3] The elements of the FAMIX model used in Evolizer can be found in Table 4.1.

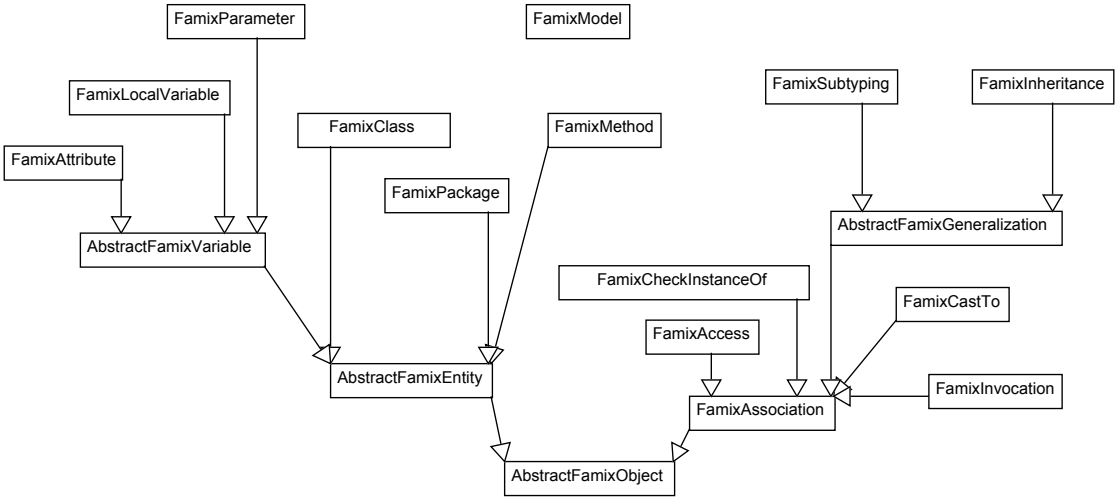


Figure 4.2: The FAMIX meta-model as implemented in Evolizer

The extracted model can be stored in a relational database, allowing access to the model through SQL queries. Alternatively, the data can be accessed through the provided Hibernate model of the Evolizer framework.

4.2.2 Model to metrics

In this thesis research, we compute most of the metrics through a combination of SQL queries and MATLAB[4] scripts. This combination allows us to quickly make changes to the metric computation, and directly process the new values in the data analysis. In contrast, using the Hibernate model would require us to recompile the metric computation code, run it, and then export the data to a statistical analysis toolkit. The disadvantage is that the FAMIX model in the database does not include all 'friendly names' for the constants and enumerations defined in the Hibernate model, but instead requires their ordinal representation.

[3] FAMIX model, http://www.moosetechnology.org/docs/famix
[4] MATLAB, http://www.mathworks.com

Entity	Description
FAMIXModel	Container holding references to the parsed entities and associations. Is the topmost entity that represents the imported project.
FAMIXPackage	The entity representing a Java package.
FAMIXClass	The entity representing a Java class. Interfaces are also represented by this entity.
FAMIXMethod	Represents a method.
FAMIXAttribute	Represents an attribute.
AbstractFAMIXVariable	Supertype of three variable types: FAMIXParameter, FAMIXLocalVariable and FAMIXAttribute
FAMIXAssociation	Supertype of all types of associations. Includes FAMIXInvocation, which is the invocation of a method. FAMIXAssociation is also a supertype for inheritance relations.

Table 4.1: Overview of FAMIX entities.

We use the Understand[5] tool for calculating several reference metrics. These include Lack of Cohesion of Methods (LCOM), Number of Methods (NOM), Lines of Code (LOC) and Weighted Method per Class (WMC). The definition of LCOM can be found in Section 3.3.2. NOM and LOC are self-explanatory. WMC is a metric proposed by Chidamber & Kemerer [6]. It is defined as the sum of the complexities of the methods within a class. Here, complexity is measured as using McCabe's cyclomatic complexity [28].

The cohesion metrics are computed using the definitions in Chapter 3, and for the interfaces and classes described in Section 4.4. This information is combined in metric computation tables, which describe how each metric is computed, and for which classes it is defined. These tables can be found in Appendix A.

4.3 Source Code Change extraction

There are several different approaches to analyze source code changes. One way is to simply count the number of revisions of a file. A disadvantage of this method is that it contains no information about the extent and nature of the changes. A slightly better approach is to count the number of Lines Modified (LM). It gives a better indication of the extent of the changes, but unfortunately not of the nature of the changes. For example, a documentation update is treated in the same way as a structural change.

[5]Understand, http://www.scitools.com/

In this research we analyze fine-grained Source Code Changes (SCC). Fine-grained SCC contain information about both the extent and the nature of the changes; they allow us to distinguish between a documentation update and the addition of a method. Using this information, we can filter out changes we consider less significant. Furthermore, previous work shows that SCC outperforms Lines Modified (LM) for learning bug prediction models [18].

To calculate the fine-grained changes, all revisions of all source code files from the versioning repository of a project have to be imported and compared. Section 4.3.1 describes how project repositories are imported, and Section 4.3.2 describes the extraction of SCC.

4.3.1 Version history importer

The Evolizer framework provides functionalities to import versioning histories of CVS, SVN and GIT projects. Evolizer can import the complete versioning history of a project, including every revision of every file and the attached log messages. The imported data is stored in a database, which data model can be seen in Figure 4.3. A *versioned file* can have one or more *revisions*, with the attached source code stored in the *content* table. *Modification reports* contain the attached log messages, with information about the commit date, author, and commit messages. Besides the elements in Figure 4.3, the data model can also store information about branches, releases, modules and transactions.

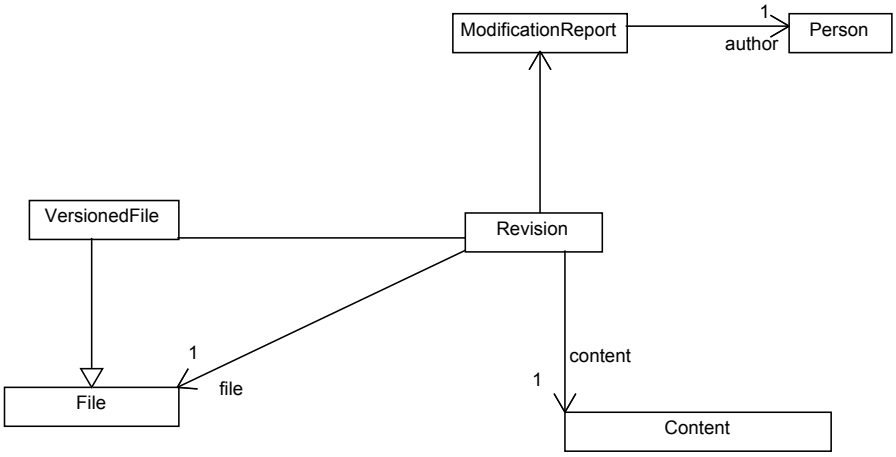


Figure 4.3: A part of the Evolizer version history importer model

The Evolizer framework has CVS, SVN and GIT versioning history importers. In this research, we have used the CVS and SVN importers. Simplified, they work as follows:

1. Import all log messages of all versions.
2. For each file, import all revisions.

During our research, we have encountered several performance limitations of especially the SVN importer. Importing large projects often took days or even weeks to complete.

We have found two methods to improve performance:

- Using the backup functionalities of SourceForge[6] and similar repositories, we were able to quickly download complete repositories.
- Custom lightweight CVS and SVN importers have been developed.

CVS projects hosted on SourceForge can be downloaded as one single file through the provided backup functionalities. This has the advantage that we do not require a connection with the Source Forge server during the entire import process. After downloading the repositories, we used a custom importer that is able to import the downloaded backups directly in the Evolizer database. More information on this importer can be found in Appendix F.

The SVN importer provided by Evolizer turned out to be quite slow. We suspect that the performance limitations are caused by the number of commands that are sent to the SVN server. The custom SVN importer has a slightly diffferent approach, reducing the number of commands issued to the SVN server.More information on the SVN importer can be found in Appendix F.

Once all data from the versioning histories is imported, we have easy access to all revisions of all source code files within the repository.

4.3.2 Change Distiller

When the complete source code history of the selected project is imported, Change Distiller [15] is used to extract the *fine-grained Source Code Changes (SCC)*. It uses a tree differencing algorithm to compare the Abstract Syntax Trees (ASTs) between all subsequent revisions of a file. This process extracts the structural changes made between two revisions, which makes it possible to distinguish between different change types.

The data extracted by the Change Distiller is stored in the database using a hibernate model, of which the class diagram can be seen in Figure 4.4. Each Java class has its own history, which contains links to each of its revisions, and all structural changes made in those revisions.

[6] http://www.sourceforge.net

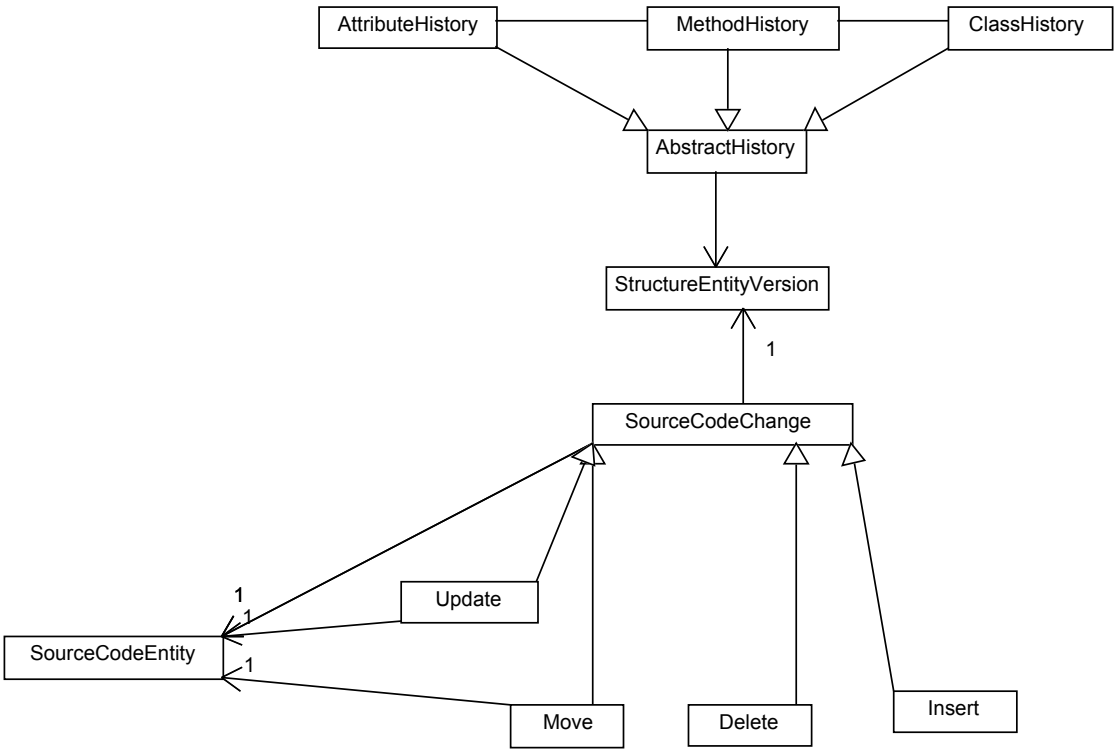


Figure 4.4: Simplified class diagram of Evolizer's Change Distiller.

In this study, we count all *significant SCC*. We consider all changes to documentation and comments to be insignificant. Appendix B presents a complete list of all changes that we consider to be significant. For the remainder of this thesis, we use the abbreviation SCC for the significant source code changes.

Although the Change Distiller is a powerful tool, there are several limitations that are worth noting:

1. If source code files are moved, it does not compare the file at the old location with the file at the new location. Instead, it considers the two files as two different entities.

2. With some complex changes, Change Distiller fails to recognize a modification operation properly. Instead, it recognizes an entity removed change, and an entity addition change.

3. The tree differencing algorithm performs badly on some very complex source files, due to enormous ASTs.

The first limitation poses a threat to the usefulness of several open source projects. Even though a project might have a sufficiently large revision history, some of the changes

cannot be used in the analysis. The second limitation has an impact on the change count for some classes, as the actual change count might be less than the calculated change count. The third limitation means that some complex source code files cannot be imported, and are excluded for analysis. These limitations are discussed in Chapter 10.

4.4 Mapping metrics to SCC

Once all metrics are computed and the number of SCC for each class has been calculated, the metrics have to be mapped to the number of SCC. Although this might seem a trivial task, there are some important decisions that can influence the results of the experiments in the next chapters.

For each class, the calculated metrics are mapped to the number of SCC through the *uniquename* of the class. The uniquename is defined in the FAMIX model, and consists of the Java package name and Java class name, separated by a dot (e.g. java.lang.Object for the Object class in the package java.lang). Inner classes are separated by a dollar sign, and anonymous classes are identified through their type and a number.

The following sections describe how the metrics are mapped to the SCC for both interfaces and concrete classes.

4.4.1 Interfaces

As described in Section 4.2.2, some of the cohesion metrics are not defined for all classes. For instance, Interface Usage Cohesion (IUC) is only defined for classes that have at least one method that is invoked by another class. If these entries are used in statistical methods, we have two options:

- Give the classes for which the metric is undefined a constant value, such as -1.
- Exclude the classes for which the metric is undefined.

The first option reduces the accuracy of the statistical methods, as obfuscating values are included, while the second option reduces the size of the dataset. Consider we want to calculate the correlation between NOM and number of fine-grained changes (SCC). We found that it makes a great difference whether we:

- Calculate NOM for all classes.
- Only include classes with at least one method.
- Only include classes with one method that is invoked by another class.

If we want to compare this correlation with the correlation of IUC and SCC, the best comparison can be made if both metrics are calculated on the same dataset. As the focus of this research is on the IUC metric, we decided to calculate as many metrics as possible on the same dataset as IUC:

- IUC, NOM,Cohesion among Methods in a Class (CAMC) and Lack of Cohesion of Interface Clients (LCOIC) are calculated for interfaces that have at least one method that is invoked by another class.
- Interface Implementation Cohesion (IIC) is calculated for the same interfaces, except the ones that do not have implementing classes.
- Normalized Hamming Distance (NHD) is calculated for classes with at least two methods.

Note that the IIC metric is undefined for interfaces that do not have an implementing class. For similar arguments as above, we could also exclude these classes from the analysis. We have decided not to do this because many of the datasets contained too few remaining entries to produce significant results.

4.4.2 Concrete classes

As described in Section 1.3, concrete classes are included in the analysis. We apply the same selection and filtering rules for concrete classes as for interfaces. However, there are a few other aspects that are worth noting.

IUC calculates the ratio of methods invoked by external classes. We know that no external class will ever invoke a private method of a class, as private methods are not visible to external classes. *We calculate IUC and LCOIC using the number of public non-static methods to handle this issue.*

This decision has at least one consequence: What is the IUC value for concrete classes that do not have any public non-static methods? We handle this in the same way as earlier in this section, *we only include concrete classes with at least one public non-static method, that is invoked at least once by another class.*

Chapter 5

Correlation Analyses

This chapter describes the investigation of the first two hypotheses. The first hypothesis defined in Chapter 1 investigates the correlation between cohesion metrics and the number of Source Code Changes (SCC). The second hypothesis investigates the correlation between cohesion metrics and the size of an interface.

5.1 Overview

As both of the hypotheses are investigated using a correlation analysis, they are investigated using the same methodology. The next section describes how the correlation analyses are performed, and how the results should be interpreted. Section 5.3 describes the investigation of the first hypothesis, and Section 5.4 describes the investigation of the second hypothesis.

5.2 Methodology

In this section we describe the methodology used in the correlation analyses. First, we describe how we calculate correlation coefficients, and how a single result is interpreted. In this research, we perform an empirical study, which means we calculate correlation coefficients for several projects. Section 5.2.2 describes how several correlation coefficients can be combined, and how we draw conclusions from these results.

5.2.1 Calculating the correlation coefficient

For both hypotheses, we calculate Spearman's rank correlation coefficient for each project. Spearman's rank correlation coefficient is a measure of statistical dependence between two variables, without assuming a specific type of relation. This is very useful when the relationship under investigation is not assumed to be linear. In both hypotheses under investigation, we have no reason to assume a linear dependency, and thus we calculate Spearman's rank correlation.

Spearman's rank correlation coefficient is a value between 0 and ±1, indicating the strength of the relationship. When the relation is negative, this is indicated by a negative sign. In our case we are only interested in the strength of the relation, and not the sign. Table 5.1 shows how the coefficients are interpreted, similar to [18].

The statistical significance of the correlation coefficients can be measured with the P-Value. Small P-values indicate that the correlation coefficient is significantly different from zero. In this research, if the P-Value of a correlation is larger than 0.05, we assume that there is no significant correlation.

Value	Type
0	No correlation
< 0.5	Weak correlation
$0.5 - 0.7$	Substantial correlation
> 0.7	Strong correlation
1	Perfect correlation

Table 5.1: Interpretation of correlation coefficients

5.2.2 Interpreting the results

After computing correlation coefficients for the selected projects, we want to make general statements about the investigated hypothesis. Having a correlation coefficient for each project under investigation, it is likely that some of the calculated correlation coefficients will support the hypothesis, while others will not.

Assume we investigate a hypothesis which claims the relation between to metrics is substantial. A possible criterion is to accept the hypothesis if the median value of all correlation coefficients is substantial (> 0.5, see Table 5.1). A maybe better criterion would be to apply a *One Sample Wilcoxon Signed-Ranks Test* on the data. This test allows us to test the correlation coefficients of a metric against a hypothesized median value. In other words, it is a measure of how likely the median value of the correlation coefficients is smaller or greater than a particular value.

We can apply the *lower-tailed One Sample Wilcoxon Signed-Ranks Test* with the null hypothesis ($H_n : Median \geq 0.5$) against the alternative hypothesis ($H_a : Median < 0.5$). If the chance that H_n is true is smaller than the significance level(0.05), then the null hypothesis is rejected in favour of H_a. In the case that H_n is accepted, this means that we have no evidence that suggest that the median is less than 0.5.

Please take note of the subtle formulation of the previous statement. If we have no evidence that the median is less than 0.5, this does not mean we have evidence that the median is greater than 0.5. To provide more solid evidence, we want to be sure that the median cannot be less than 0.5. To test this, we can perform an *upper-tailed* One Sample Wilcoxon Signed-Ranks Test with the null hypothesis ($H_n : Median \leq 0.5$) against the alternative hypothesis ($H_a : Median > 0.5$). If H_n is rejected we can be confident that the median is at least 0.5. To summarize, we have three possible output scenarios:

1. H_n : $Median \geq 0.5$ is rejected in favor of H_a : $Median < 0.5$: The median correlation is very unlikely substantial.

2. H_n : $Median \leq 0.5$ is rejected in favor of H_a : $Median > 0.5$: The median correlation is very unlikely not substantial.

3. Both tests accept the null hypothesis: We cannot exclude that the median correlation is substantial, nor can we exclude that it is weak.

There is one more thing worth noting about the correlation analysis. A strong correlation should not be confused with causality. Neither does a correlation coefficient provide information about the type of relation. We provide scatter plots and data tables to provide information about the distribution of the values. Based on these results, we can then decide to accept or reject a hypothesis.

5.3 Correlation analysis between metrics and changes

We apply the methodology described in the previous section to investigate the first hypothesis. Recall the definition of the first hypothesis from Chapter 1:

> H_1: The cohesion metrics are correlated with the number of fine-grained changes in Java classes.

The goal of this hypothesis is to investigate the relation between the cohesion metrics and the number of SCC. We investigate this for each metric independently, both for interfaces and concrete classes. The process below is executed for each metric under investigation:

1. For each project: calculate Spearman's coefficient between the cohesion metric and the number of SCC.

2. Check for each project whether the correlation is significant and substantial according to the definitions.

To accept H_1 for a cohesion metric, we have to be confident that the cohesion metric is substantially correlated with the number of SCC. To do this, we follow the methodology described in the previous section. Based on the median correlation coefficient, the results of the Wilcoxon tests and the supporting scatter plots and tables, we can either accept or reject the hypothesis.The same process is repeated for concrete classes. It is interesting to see whether the cohesion metrics correlate differently with SCC for concrete classes.

5.4 Correlation analysis between metrics and size

Similar to the investigation of the first hypothesis, we investigate the second hypothesis:

> H_2: The cohesion metrics are correlated with the size of Java classes.

We investigate the relation between cohesion metrics and size metrics, to see if possible correlations between metrics and changes can be caused by size. In other words, if a cohesion metric correlates with the number of changes, is this because a lack of cohesion causes more changes, or because larger classes tend to be less cohesive? The interface size metric we investigate is Number of Methods (NOM). For concrete classes, we include two different size metrics: Lines of Code (LOC) and NOM.

This study aims to find proof to accept H_2. H_2 can be accepted if there is a substantial correlation between the cohesion metrics and the size metrics. Again, *One Sample Wilcoxon Signed-Ranks Tests* can be performed to test the correlation coefficients against a hypothesized median value. Based on the results of the Wilcoxon tests and the supporting data, H_2 can be accepted or rejected for a metric.

Chapter 6

Prediction Models

The previous chapter described the approach to the investigation of the first two hypotheses. This chapter describes the approach to the investigation of the third hypothesis, which is defined as follows:

> H_3: The cohesion metrics can improve the performance of prediction models to classify Java classes into change- and not change-prone.

6.1 Overview

In this chapter we describe the methodology applied to assess whether cohesion metrics can improve change prediction models. A change prediction model is a model that is able classify interfaces as being *change-prone* or as *not change-prone*, based on several input variables.

To explain some of the basic concepts and terminology used in this chapter, Section 6.2 gives a short introduction to classification models.

We found that our initial experimental setup has several underlying assumptions. To perform a thorough analysis, we have designed three different experiments. Section 6.3 describes the common characteristics of these experiments. We consider the entire process of constructing one or more prediction models and the comparison of performances as a single experiment.

The selected experiments are described in Section 6.4. Finally, the last section presents a summary of the chapter.

6.2 Introduction to classification models

Classification is defined as a systematic arrangement in groups according to established criteria.[1] In the field of machine learning, classification models are often constructed to automatically create such an arrangement. To explain the use of classification models in this thesis, we start with a small example.

[1] http://www.merriam-webster.com/dictionary/classification

Assume we have a dataset D of a single software project consisting of several Java classes and interfaces. For each interface i in D, we have gathered the following information:

- The name of the interface N_i.
- A set of metric values X_i.
- The number of changes of the interface C_i.

The goal is to construct a prediction model M that is able to predict whether an interface i is change-prone or not using the metric values X_i. The process is illustrated in Figure 6.1. First we convert the number of changes of each interface to a label y_i. Based on the number of changes C_i and the definition of change-proneness, $y_i = 1$ (change-prone) or $y_i = 0$ (not change-prone). Then, the dataset is split in a training set D_{train} and a test set D_{test}.

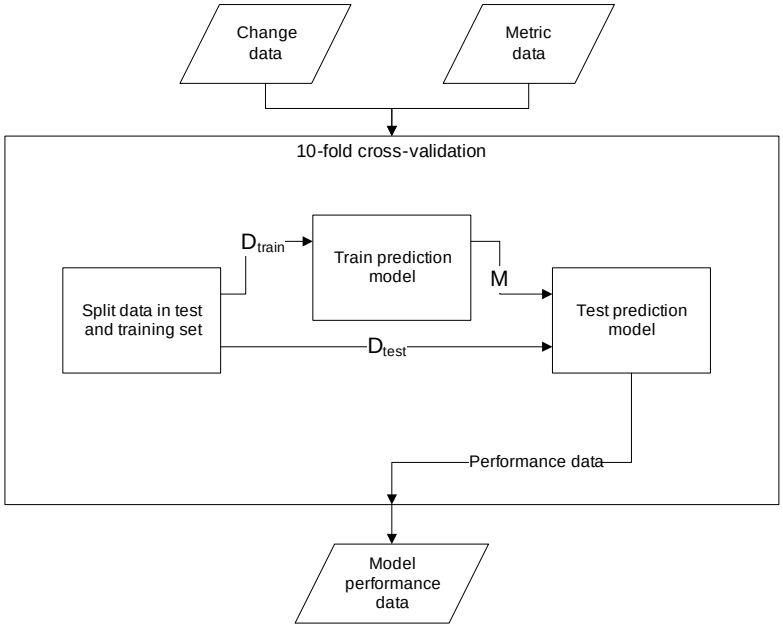


Figure 6.1: The prediction model framework

A selected classification algorithm is applied on the training set D_{train}, to train a model which is able to predict a label $\hat{y}_i$ using the metrics X_i for any interface in D. In other words, we are generating a function $\hat{y}_i = M(X_i)$.

Note the distinction between the actual label y_i and the predicted label $\hat{y}_i$. Applying the classification model on the test set D_{test} results in a predicted value $\hat{y}_i$ for each interface $i \in D_{test}$. This prediction indicates whether the interface is *expected* to be change-prone or not change-prone.

Once all labels are predicted for the interfaces in the test set, we can compare the actual labels y_i with the predicted labels $\hat{y}_i$. This comparison can be used to compute the performance of the classification model, making it possible to compare different models. This is discussed in more detail in Section 6.3.5.

The process of training and testing is often repeated K times with different subsets of the data. This process is called cross-validation, and is used to reduce the variability of the dataset. In this research, we apply 10-fold cross-validation.

6.3 Construction of a single experiment

In this section we describe how a single experiment is constructed. A single experiment is the complete process that can be used to assess whether cohesion metrics can improve prediction models, but then for a specific configuration. This section describes how each experiment is constructed, and the steps out of which an experiment consists:

1. Dataset selection.
2. Definition of change-proneness.
3. Construction of prediction models.
4. Selection of classification algorithms.
5. Definition of performance criteria to compare the prediction models.
6. Application of prediction models.

6.3.1 Dataset selection

Similar to the correlation analyses, we have to select a model on which we compute the source code metrics, and period of time on which we analyze the changes. We investigate the correlation between the cohesion metrics of the most recent version of the project and all available changes that have been made in the past. The advantage of this method is that we have as much data as possible to analyze. The disadvantage is that we assume that classes will change in the future the same way as they have changed in the past.

An idea is to select different releases of a project, and assess whether a project can predict future changes based on metrics computed with an earlier release. In Section 6.4 we will describe experiments that take into account different releases and change sets.

6.3.2 Definition of change-proneness

Similar to [36], an interface is considered change-prone if the number of changes of that interface exceeds the median value for that project. A more formal definition:

$$interface = \begin{cases} \text{change-prone} & if \#SCC > median \\ \text{not change-prone} & else \end{cases}$$

Note here that we choose the median as the cut-off point. A valid question is whether another cut-off point will provide different results. Due to limited time, we could only do limited research with different cut-off points. Section 6.4 describes experiments with different cut-off points.

6.3.3 Construction of prediction models

The experiments described in this chapter are performed by comparing classification models using different metric sets as input. The baseline model for this study is a prediction model based on size metrics. The expectation is that larger classes are likely to have more changes, and are thus more often change-prone than small classes.

Recall the size metrics used in the correlation analysis (Chapter 5): NOM for interfaces; NOM, LOC and Weighted Method per Class (WMC) for concrete classes. We define the input set for the baseline model as $X_i = \{NOM\}$ for interfaces, and the input set for the baseline model for concrete classes as $X_i = \{NOM, WMC, LOC\}$.

Based on the constructed baseline model, we can construct at least one extended model based on cohesion metrics. The first model we construct is an extended model using all size and cohesion metrics. Furthermore, we create a second alternative model based on the performed correlation analysis. All cohesion metrics that produce sufficient significant results are included in that model. All constructed models can be seen in Table 6.1.

Model	Input metrics
Baseline	Size metrics
All	All size and cohesion metrics
Promising	Size metrics and promising cohesion metrics

Table 6.1: Baseline and extended prediction model

6.3.4 Selection of classification algorithms

Different algorithms can be used for classification purposes. In this research, we have selected several algorithms. They all work as described earlier in this section, and illustrated in Figure 6.1. The following paragraphs describe the algorithms in more detail.

Logistic Regression Logistic regression is a type of model that is able to predict the probability that an instance belongs to a certain class, based on several predictor variables. For example, it can be used to determine the probability that a person will get a certain disease based on his age, sex and other variables. As the name indicates, logistic regression is related to linear regression. A typical linear regression function looks as follows:

$$f(X) = \beta_0 + \beta_1 x_1 + \beta_2 x_2$$

The function above tries to estimate the value of y_i with the predictive function $f(X)$. It multiplies the input variables $X = \{x_1, x_2, ...\}$ by the calculated regression coefficients β_i to estimate y. β_0 is the intercept. By transforming the linear regression function to a logistic function, the output values lie within the range 0-1. This way it can be used as a binomial classification algorithm. An example:

$$z = \beta_0 + \beta_1 x_1 + \beta_2 x_2$$

$$f(z) = \frac{1}{1 + \exp^{-z}}$$

When the classification algorithm is trained, the coefficients β_i are calculated. A larger coefficient β_i indicates a stronger relation between the input variable x_i and the predicted variable y.

Decision Tree Although the input and output variables are similar, the decision tree algorithm is a quite different from logistic regression. A decision tree can be described as a set of rules that is used to predict an output value(e.g. if $X > 5$ then $Y = A$ else $Y = B$). There are several algorithms that can be used to generate a decision tree based on training data. Usually, these algorithms consist of *grow* and *prune* operations. A tree is grown when new splits are added to the tree. Pruning is the act of removing nodes from the tree whenever they do not provide sufficient new information. This is often done to avoid overfitting the model.

Random forest The random forest algorithm is a variation of the decision tree learning. The idea is to grow N decision trees. For each tree, a random subset of input variables is selected to construct the tree. All trees can be grown using the training dataset. When the generated model is applied on the test data, the algorithm decides a classification based on a voting model. For example, if 8 out of 10 decision trees predict the label A, the algorithm will very likely select A as the final prediction. There are several different criteria that can be used for growing the trees, selecting input variables and voting models.

Naive Bayes The Naive Bayes (NB) algorithm is a classifier that is based on Bayes theorem. The algorithm 'Naively' assumes independence of the input parameters, and uses them to predict a class for an entity.

Support Vector Machine The Support Vector Machine (SVM) algorithm creates a model that represents points in space. It classifies by mapping an entity in that space, and then checking to which class it is closest.

Neural Network The Neural Network (NN) algorithm is an algorithm inspired by the biological neural network, and uses neurons and weights to classify an entity.

6.3.5 Definition of performance criteria to compare prediction models

To test whether cohesion metrics can improve prediction models, objective performance criteria have to be defined to compare prediction models. The criteria should indicate how accurate the prediction models are. Based on previous work [18, 36], we have selected the following performance metrics to compare the different classification models: Precision and recall, F-measure and area under the curve (AUC). Furthermore, we have selected Matthews Correlation Coefficient (MCC) based on research by Baldi [3]. These metrics provide us with information about the differences between the classification models, and are explained in more detail in this section.

Precision, Recall and the F-Measure There are several performance metrics that can be used to compare classifcation models. Table 6.2 gives an overview of the possible outcomes of a prediction change-prone (CP) or not change-prone (NCP). Besides the correctly predicted True Positive (TP) and True Negative (TN), there are two types of incorrect predictions: False Positive (FP) and False Negative (FN). These errors can be used to calculate several performance metrics, such as *precision* and *recall*.

		Class	
		CP	NCP
Predicted Class	CP	TP	FP
	NCP	FN	TN

Table 6.2: Possible classification outcomes.

$$precision = \frac{TP}{TP + FP}$$

$$recall = \frac{TP}{TP + FN}$$

Related to precision and recall is the *F-measure*[42], which is defined as follows:

$$F = 2 \cdot \frac{recall \cdot precision}{recall + precision}$$

Matthews Correlation Coefficient In machine learning, the MCC can be used to measure the quality of a binary classification [3]. It takes into account all values of the contingency table, where the F-Measure does not include the true negatives. An interesting property is that it does not care about which class is positive, and which

class is negative. In other words, if we invert the two classes, the F-Measure would get a different value, where the MCC value remains the same. It is defined as follows:

$$\text{MCC} = \frac{TP \cdot TN - FP \cdot FN}{\sqrt{(TP+FP)(TP+FN)(TN+FP)(TN+FN)}}$$

ROC Curves Receiver Operating Characteristic (ROC) curves are a more graphical method of comparing the performance of different classification models. It is a plot of the True Positive (TP) rate versus the False Positive (FP) rate, which are defined as: $tpr = \frac{TP}{TP+FN} \times 100\%$ and $fpr = \frac{FP}{FP+TN} \times 100\%$.The diagonal line of an ROC curve represents a random classifier. Any classifier below that line performs worse than a random guess. If the output of that classifier is inverted it becomes a good classifier.

A single classifier can be represented by a point in the ROC curve. By varying the discrimination threshold, a curve can be constructed for a classifier. For example, a logistic regression classifier could classify a sample as A whenever $P(X = A) \geq 0.5$. Varying this threshold (0.5) can increase the TP rate, while decreasing the FP rate, and vice versa.

ROC curves can be used to compare different classification models. The AUC can be calculated to compare different classifiers. The AUC gives an indication which classifier generally performs better. It is argued [42] that AUC provides very limited information to compare classifiers. For example, it does not distinguish between classifiers tending to provide a higher TP rate and classifiers that provide a lower FP rate. Depending on the type of research, a lower FP rate could be more valuable than a higher TP rate.

6.3.6 Application of the prediction models

Combining the information from the previous sections, we can fill in the framework illustrated in Figure 6.1. The figure illustrates the cycle we perform for a single classification model. The output are the 4 selected performance metrics: Precision and Recall, F-Measure and the area under the curve (AUC) of the ROC.

For each input model (baseline and extended), we train and test the three selected classification models. This is repeated for each project, which means we have 3 pairs of models per project. The models are trained and tested using RapidMiner.[2] Information about the configuration of the different models can be found in Appendix C. Ten-fold cross-validation is applied to the models in order to reduce the variability.

Finally, the output metrics of the two different input models have to be compared. Similar to the correlation analysis, we can use the *Wilcoxon Signed-Ranks Test* to see whether the models produce significantly different results. In order to accept H_3, we have to be confident that the extended model is significantly better than the baseline model.

[2]RapidMiner http://www.rapidminer.com

6.4 Selection of experiments

The previous section describes many different options for constructing prediction models, selecting datasets and the definition of change-proneness. To perform a thorough investigation of the hypothesis, it is necessary to investigate several different configurations. As it is nearly impossible to try all configurations, we have selected several experiments. Table 6.3 shows an overview of the selected experiments, and they are described in the following sections.

Experiment	Goal
Core experiment	Base experiment to investigate the hypothesis.
Different change-prone cutoff points	Do different cutoff points for the change-prone definition lead to better prediction models?
Predict between releases	Investigate H_3 between releases of a software project.

Table 6.3: The selection of experiments.

6.4.1 Core experiments

The core experiments have the following characteristics:

- Metrics are computed on the latest version of the source code.
- Changes are computed over the complete versioning history.
- The median of the number of SCC is selected as cutoff point for the definition of change-prone. (see Section 6.3.2)

The core experiments are executed for both interfaces and concrete classes separately.

6.4.2 Different change-prone cutoff points

As described in Section 6.3.2, taking the median value of the number of SCC is just one cutoff point. In this experiment, we investigate whether the results are different if we use different cutoff points. We perform the experiment with the following alternative cutoff points:

- Respectively the top 25%, the top 10% and top 5% of all classes is considered change-prone.
- All classes with at least one change are considered change-prone.

6.4.3 Predicting between releases

The core experiments investigate the capabilities of source code metrics to predict the number of SCC that have been made in the past. The actual goal of this entire investigation is to predict future changes, where the core experiments can explain changes in the past at best. Many software projects have different releases throughout their history. We can calculate the source code metrics on a release R_1, and see whether these metrics can be used to predict the changes between R_1 and R_2.

Part III

Empirical study

Chapter 7

Project Selection

This chapter describes the selection of the projects for the empirical study. Using these projects, the analyses and experiments in the previous chapters can be executed.

7.1 Overview

The next section of this chapter describes the basic requirements for the project selection. After that, the final section describes the projects selected for the empirical study. In that section we also analyze the composition of the projects in terms of versioning history and software structure.

7.2 Project requirements

As previous research [36] focused primarily on projects from the Eclipse and Hibernate frameworks, this empirical study includes both standalone and framework open source projects. We have located several projects using the open source project repositories of Eclipse, Apache and SourceForge.[1]

In order to assure that the results of the analysis are statistically significant, we found that a project should meet the following requirements:

- Projects should be of sufficient size, preferably more than 500 classes.
- The project should have a long revision history, with at least 2000 revisions.

Note that these requirements give no guarantee for statistical significant results, but it is a good starting point for the project selection. During our selection, we found that projects with less data did not produce significant results. In some cases we noticed that the limitations of the research framework (see Chapter 4) resulted in less usable projects.

[1] www.eclipse.org, www.apache.org, www.sourceforge.net

7.3 Selected projects

The final selection can be found in Table 7.1. For each project, it shows the following statistics:

- The number of interfaces and concrete classes (CClasses) measured at the latest version.
- The dates between which all file revisions are gathered and the number of revisions between those dates.
- The number of significant fine-grained Source Code Changes (SCC) (see Section 4.3.2).

Project	Interfaces	CClasses	From	To	File Revisions	SCC
ArgoUML-app	108	1,598	2/2008	10/2011	6,811	21,876
Eclipse.debug.ui	136	1,259	5/2001	9/2011	22,468	75,388
Eclipse.jdt.core	196	1,194	6/2001	9/2011	80,303	401,415
Eclipse.team.core	48	177	11/2001	5/2011	4,625	6,600
Hibernate3	205	1,508	6/2004	2/2006	12,456	49,019
Jabref	39	1,395	10/2003	10/2011	6,919	53,140
JEdit	63	1,065	9/2001	10/2011	10,708	90,001
Jena2	204	2,062	12/2002	5/2011	30,298	91,713
JFreechart	85	682	8/2004	7/2007	8,076	20,576
Joone	36	481	3/2001	10/2008	5,194	21,224
Log4J	23	413	12/2000	9/2011	2,831	10,218
Lucene	44	973	9/2001	11/2011	19,655	36,600
Rapidminer	205	4,045	1/2010	10/2011	11,165	20,613
Sweet Home 3D	27	1,395	11/2005	9/2011	22,445	45,924
TripleA	104	1,812	1/2002	11/2011	8,753	22,453
Vuze	1,054	6,391	7/2003	4/2010	54,713	376,910
Xerces	160	914	11/1999	9/2011	14,871	117,462

Table 7.1: The project selection for the empirical study,

We observe that some statistics vary greatly per project, such as the interfaces/CClasses ratio and the number of significant changes per revision. This is an indication that the project selection is diverse, and it might be interesting to look at the differences between the projects. In the next section the differences between the distributions of changes is analyzed in more detail.

7.3.1 Project change distribution

Table 7.2 shows several statistics about the change sets of the interfaces for the different projects. Note the distinction between interfaces that have 0 SCC and interfaces that do have significant changes. These statistics will influence the analyses in the next sections.

Table 7.3 shows the same statistics for the concrete classes. Observe that Jena2 has around 48,000 changes linked to the concrete classes and interfaces, while Table 7.1

Project	Size	Max	SumChanges	Mean	NonZero	Median($\neq 0$)
Argouml-app	108	28	76	0.70	11 (10%)	3
Eclipse.debug.ui	136	270	1014	7.46	58 (43%)	4
Eclipse.jdt.core	196	785	2753	14.05	110 (56%)	6
Eclipse.team.core	48	30	171	3.56	23 (48%)	3
Hibernate3	205	346	1610	7.85	96 (47%)	5
Jabref	39	23	60	1.54	15 (38%)	1
JEdit	63	24	100	1.59	20 (32%)	3.5
Jena2	204	187	2005	9.83	117 (57%)	5
JFreeChart	85	51	145	1.71	18 (21%)	3
Joone	36	25	117	3.25	18 (50%)	4
Log4J	23	65	125	5.43	10 (43%)	7.5
Lucene	44	25	65	1.48	10 (23%)	3.5
RapidMiner	205	13	83	0.40	28 (14%)	1.5
Sweet3D	27	56	142	5.26	10 (37%)	4
TripleA	104	54	295	2.84	42 (40%)	3
Vuze	1054	701	8994	8.53	546 (52%)	5
Xerces	160	56	454	2.84	63 (39%)	2

Table 7.2: Statistics of the SCC of the interfaces.

Project	Count	Max	SumChanges	Mean	NonZero	Median($\neq 0$)
Argouml-app	1598	367	16764	10.49	486 (30%)	12
Eclipse.debug.ui	1259	3189	44320	35.20	465 (37%)	23
Eclipse.jdt.core	1194	27612	342287	286.67	845 (71%)	59
Eclipse.team.core	177	471	2131	12.04	65 (37%)	15
Hibernate3	1508	2297	26456	17.54	530 (35%)	17
Jabref	1395	1772	45741	32.79	404 (29%)	30.5
JEdit	1065	6123	60719	57.01	301 (28%)	46
Jena2	2062	4188	46773	22.68	793 (38%)	17
JFreeChart	682	783	18160	26.63	504 (74%)	10
Joone	481	976	16414	34.12	248 (52%)	27.5
Log4J	414	680	8741	21.11	144 (35%)	21
Lucene	973	1124	11292	11.61	318 (33%)	10
Rapidminer	4045	637	17508	4.33	887 (22%)	5
Sweet3D	1395	4200	32974	23.64	151 (11%)	73
TripleA	1812	1229	17271	9.53	260 (14%)	24
Vuze	6391	7012	247181	38.68	1685 (26%)	28
Xerces	914	3510	62253	68.11	543 (59%)	18

Table 7.3: Statistics of the SCC of the concrete classes.

shows that the project should have around 91,000 changes. Even when we include abstract classes, about 40% of all changes remain unlinked. This is a perfect illustration of one of the limitations of the research framework. Apparently, much of the source code has been moved, which means Change Distiller is unable to relate these changes to the latest version of the source code.

Another interesting project is RapidMiner. The linked changes for interfaces and concrete classes are in total close to the total number of imported changes (20,613), which means the imported history can actually be used. When we compare RapidMiner to Log4J, we see that RapidMiner has about 10 times the number of interfaces Log4J has, while the total number of interface changes is higher for Log4J. This can be explained by the fact that the imported RapidMiner history is relatively short, only from 2010-2011, where Log4J has a history of over 10 years.

Concluding, we have selected a variety of open source projects for the empirical study. Even though the projects adhere to the minimal requirements for selection, some of the projects have limited data available. This is something that should be considered in the statistical analyses.

Chapter 8

Results of the Correlation Analyses

In this chapter we investigate the first two hypotheses defined in Section 1.3:

> H_1: The cohesion metrics are correlated with the number of fine-grained changes in Java classes.
> H_2: The cohesion metrics are correlated with the size of Java classes.

The previous chapters described the research framework and the process designed to investigate these hypotheses. Chapter 5 described how the correlation analyses are performed, and how the results should be interpreted. This chapter describes the execution of the correlation analyses using the open-source projects selected in Chapter 7.

8.1 Overview

As described in Chapter 5, the first two hypotheses are investigated by performing a correlation analysis. Each of the two investigations follows the following structure:

1. **Results:** the direct results from the study.
2. **Analysis:** this section analyzes the results for each metric.
3. **Conclusion:** evaluate the hypothesis under investigation.

Section 8.2 describes the results of the correlation analysis between the cohesion metrics and the number of changes. To analyze the effects of size, in Section 8.3 the relation between cohesion and size metrics is investigated. In each section the hypotheses are investigated primarily for the Java interfaces. The concrete classes are described in the same sections, and are included for support. This chapter ends with a summary of the results in Section 8.4.

8.2 Correlation analysis between cohesion metrics and SCC

We investigate the relation between interface cohesion metrics and the number of fine-grained Source Code Changes (SCC) by performing a correlation analysis. In this section, we try to find evidence to accept or reject the first hypothesis:

> H_1: The cohesion metrics are correlated with the number of fine-grained changes in Java classes.

8.2.1 Results

Table 8.1 shows the results of the correlation analysis between the cohesion metrics and the number of SCC of Java interfaces. Recall from Chapter 5 that a correlation of -1 and 1 indicate very strong correlations, and a correlation of 0 indicates no correlation. The metric that scores best for a single project is highlighted in bold.

Note that the number of interfaces under analysis is smaller than the available number of interfaces in the project. This is due to the fact that interfaces without methods or clients are excluded from the analysis, as is described in Section 4.4.

Project	NOM	IUC	CAMC	NHD	IIC	LCOIC
Argouml-app	0.19	-0.27*	**-0.32****	-0.03	-0.22	0.24*
Eclipse.debug.ui	0.43**	**-0.52****	-0.40**	0.11	-0.13	0.15
Eclipse.jdt.core	**0.59****	-0.48**	-0.46**	0.31*	-0.35**	0.26**
Eclipse.team.core	0.44**	-0.44**	-0.22	0.29	-0.17	**0.53****
Hibernate3	**0.61****	-0.60**	-0.51**	0.43**	-0.16*	0.40**
Jabref	0.67**	**-0.69****	-0.31	0.42	-0.58**	0.66**
JEdit	0.22	0.04	-0.20	0.16	-0.22	0.15
Jena2	**0.68****	-0.64**	-0.61**	0.37**	0.05	0.46**
JFreeChart	0.15	-0.01	-0.17	0.13	-0.17	-0.04
Joone	**0.69****	-0.51**	-0.59**	0.55*	-0.35	-0.05
Log4J	0.52*	-0.49*	-0.67**	**0.89***	-0.31	-0.01
Lucene	**0.60****	-0.45**	-0.46**	0.53**	-0.20	0.31*
RapidMiner	0.34**	-0.35**	-0.32**	0.17	-0.07	**0.42****
Sweet3D	0.57**	-0.56**	**-0.61****	0.48	-0.34	0.33
TripleA	**0.51****	-0.48**	-0.41**	0.43**	-0.39**	0.28**
Vuze	**0.74****	-0.67**	-0.62**	0.35**	-0.28**	0.51**
Xerces	**0.48****	-0.47**	-0.34**	0.24*	-0.15	0.32**

Table 8.1: Spearman's rank correlation between number of SCC and cohesion metrics and NOM for interfaces. (* indicates a significant correlation at $\alpha = 0.05$, ** at $\alpha = 0.01$)

Table 8.2 shows the correlation analysis of cohesion metrics and SCC for concrete classes, and Table 8.3 shows the similar correlations for four other OO metrics that are only defined for classes.

Note the distinction between the Number of Public Methods (NOPM) and Number of Methods (NOM). This distinction is made to see the differences between the two

metrics, as Interface Usage Cohesion (IUC) is only calculated for all non-static public methods. This will be described in more detail in Section 8.3.

Project	NOPM	IUC	CAMC	NHD	LCOIC
ArgoUML-app	0.32**	-0.32**	-0.45**	0.43**	0.20**
Eclipse.debug.ui	0.32**	-0.27**	**-0.56****	0.30**	0.28**
Eclipse.jdt.core	0.52**	-0.45**	**-0.63****	0.56**	0.32**
Eclipse.Team.core	0.56**	-0.31**	**-0.64****	0.58**	0.24*
Hibernate3	0.20**	**-0.36****	-0.26**	0.19**	0.28**
Jabref	0.47**	-0.38**	**-0.51****	0.40**	0.23**
JEdit	0.32**	-0.25**	-0.34**	**0.35****	0.18**
Jena2	0.54**	-0.41**	**-0.56****	0.40**	0.17**
JFreeChart	0.58**	-0.39**	**-0.62****	0.49**	0.14*
Joone	0.53**	-0.47**	**-0.54****	0.51**	0.19**
Log4J	0.47**	-0.30**	**-0.49****	0.41**	0.24**
Lucene	0.40**	-0.31**	**-0.43****	0.37**	0.22**
RapidMiner	0.28**	-0.28**	**-0.33****	0.28**	0.23**
Sweet3D	0.48**	-0.52**	**-0.60****	0.37**	0.49**
TripleA	0.27**	-0.26**	**-0.36****	0.29**	0.26**
Vuze	0.47**	-0.44**	**-0.54****	0.27**	0.31**
Xerces	0.38**	-0.29**	**-0.52****	0.30**	0.22**

Table 8.2: Spearman's rank correlation between the number of SCC and cohesion metrics and NOPM for concrete classes. (* indicates a significant correlation at $\alpha = 0.05$, ** at $\alpha = 0.01$)

Appendix D contains additional results in the form of tables and scatter plots, which we can use for further analysis. Section D.2 contains scatter plots of the cohesion metrics versus SCC, to support the correlation coefficients. Section D.1 contains information about the distribution of the metrics.

8.2.2 Analysis

The correlation data shows interesting results, which are summarized in Tables 8.4 and 8.5. The last column of those tables indicates the results of a *One Sample Wilcoxon Signed-Ranks Test* on the correlation coefficients. As explained in Chapter 4, the *One Sample Wilcoxon Signed-Ranks Test* tests the null-hypothesis that the median of a collection of values is equal to a specified value, against the alternative that the median is not equal, less, or greater than the specified value. Recall from the previous chapter that we consider a correlation between 0.5 and 0.7 to be substantial. The column 'IsSubstantial' is the result of the two Wilcoxon tests as described in Section 5.2.2, and tests the null hypothesis $median(correlation) = 0.5$. The significance level of the Wilcoxon tests is 0.05.

Project	NOM	LOC	WMC	LCOM
ArgoUML-app	0.47**	**0.54****	0.49**	0.27**
Eclipse.debug.ui	0.60**	0.71**	**0.72****	0.53**
Eclipse.jdt.core	0.67**	**0.83****	0.82**	0.43**
Eclipse.Team.core	**0.70****	0.64**	0.66**	0.39**
Hibernate3	0.28**	**0.54****	0.45**	0.08*
Jabref	0.47**	**0.65****	0.59**	0.43**
JEdit	0.38**	**0.54****	0.43**	0.30**
Jena2	0.68**	0.68**	**0.70****	0.41**
JFreeChart	0.65**	**0.77****	**0.77****	0.34**
Joone	0.53**	0.53**	**0.55****	0.39**
Log4J	0.39**	**0.47****	**0.47****	0.16
Lucene	0.37**	**0.43****	0.42**	0.32**
RapidMiner	0.38**	**0.43****	0.38**	0.28**
Sweet3d	0.57**	**0.83****	0.75**	0.46**
TripleA	**0.38****	0.36**	0.35**	0.31**
Vuze	0.39**	0.55**	0.45**	**0.66****
Xerces	0.55**	**0.71****	0.70**	0.41**

Table 8.3: Spearman's rank correlation between the number of SCC and reference metrics for concrete classes. (* indicates a significant correlation at $\alpha = 0.05$, ** at $\alpha = 0.01$)

Metric	Significant	Substantial	Median	IsSubstantial
IUC	15	7	-0.48	Possibly
CAMC	13	6	-0.41	No
LCOIC	11	3	0.31	No
NHD	9	3	-0.35	No
IIC	5	2	0.22	No

Table 8.4: Summary of the correlation between SCC and cohesion metrics for interfaces.

Metric	Significant	Substantial	Median	IsSubstantial
CAMC	17	10	-0.52	Possibly
NHD	17	3	0.37	No
IUC	17	1	-0.32	No
LCOIC	17	0	0.23	No

Table 8.5: Summary of the correlation between SCC and cohesion metrics for concrete classes.

General observations

For interfaces, the projects Argouml-App, JEdit and JFreechart show few significant correlations. This can be partially explained by a lack of data in the projects, which emphasizes the need for large projects with many interfaces and many significant changes. Note that these projects are not the smallest projects in terms of the number of interfaces and classes. Some smaller projects do have significant correlations with the cohesion metrics, such as Joone, Log4j and Sweet Home 3D.

Another interesting project is RapidMiner: it has relatively many interfaces, but has few substantial correlations with the metrics. Table 7.2 shows that only 1 out of 10 classes has one or more significant SCC. Figure 8.1.2 shows that RapidMiner has few unique data points in the plot of Interface Implementation Cohesion (IIC) and SCC. This makes it difficult to draw conclusions about a correlation. In Appendix D, similar results are found for other metrics and other projects.

For concrete classes we see more significant correlations. This can primarily be explained by the size of the datasets; most projects have significantly more concrete classes than interfaces. Furthermore, we have seen in Chapter 7 (see Table 7.3) that each project has over 100 concrete classes with at least one SCC.

The second observation for concrete classes is that the correlation coefficients for most of the cohesion metrics are lower than those of the interfaces. We think that there are at least two explanations for this.

First of all, concrete classes contain more information than interfaces. Where interfaces primarily contain method declarations, concrete classes also contain attributes and method bodies. These extra elements give concrete classes more reasons for change. As cohesion is another reason for change, we think the ratio of the cohesion related changes versus all changes is much less for concrete classes than for interfaces.

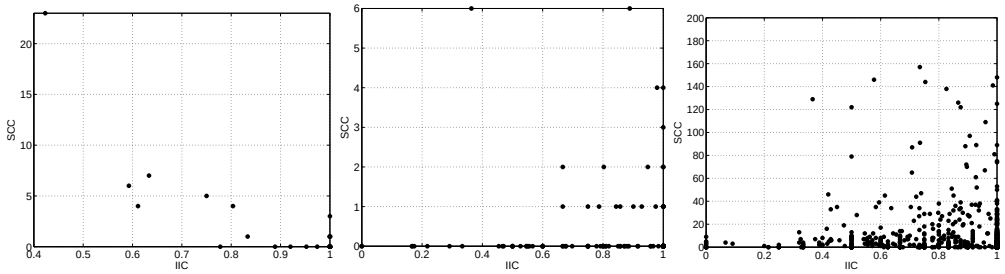
Another explanation for the low correlation coefficients lies in the definition of the cohesion metrics. For instance, we calculate IUC only over public non-static methods, which means it is only calculated over a part of the class. This means that IUC ignores the private and protected methods, where changes will be made as well. The next sections analyze the results per metric.

Interface Implementation Cohesion

In general, IIC shows weak correlations with the number of changes. The correlations are often insignificant as well. Our analysis finds multiple causes:

- As indicated in Section 4.4, some interfaces do not have any implementing classes. By excluding these cases in the analysis, the dataset becomes smaller which results in even less data to analyze.

- IIC=1 for many interfaces (see Appendix D), which indicates that many interfaces are completely implemented. These interfaces often do have changes, and thus have an impact on the correlation. In fact, for each project at least half of all interfaces has an IIC value of 1.



1: JabRef 2: RapidMiner 3: Vuze

Figure 8.1: Scatter plots that show the relation between IIC and SCC for interfaces of three projects.

IIC has only one substantial correlation with the number of SCC, for the project Jabref. Looking at the scatter plot in Figure 8.1.1, we can see that this correlation is based on few data points, and thus may be less reliable.

Analyzing the scatter plots of larger projects shows us something interesting. Most of the correlation coefficients are negative, which indicates that an interface with a lower IIC value will likely have more changes. The scatter plots seem to show more of a relation in the other direction, as can be seen in Figure 8.1.3. Interfaces with many changes seem to have higher IIC values, which seems to be the case for Vuze.

To explain this phenomenon, we observe that the scatter plots give only insight in the unique data points, and the correlation values can be explained by the fact that there are very much interfaces with high IIC values.

Usage Cohesion

IUC and Lack of Cohesion of Interface Clients (LCOIC) are the two usage cohesion metrics we investigate. Tables 8.1 and 8.4 show that IUC has more significant and more substantial correlations with the number of changes than LCOIC.

In Tables 8.4 and 8.5 it can be seen that the median values for LCOIC are low. Based on the *One Sample Wilcoxon Signed-Ranks Tests* the hypothesis that the correlation is substantial can be rejected. The scatter plots and distibution tables in Appendix D show that LCOIC has many 0 values, and support the conclusion that LCOIC does not seem to be substantially correlated with SCC.

IUC shows stronger correlations with both interfaces and concrete classes. For interfaces, the median is 0.48 and the *One Sample Wilcoxon Signed-Ranks Test* accepts the null hypothesis $Median(LCOIC) \geq 0.5$ in favor of the alternative $Median(LCOIC) < 0.5$ at a significance level of 0.05. Similarly the Wilcoxon tests accept the null hypothesis $Median(LCOIC) \leq 0.5$ in favor of the alternative $Median(LCOIC) > 0.5$. The first test implies that the median can be substantial, where the second test implies the median does not have to be substantial, as described in Chapter 5.

Further investigations of IUC show that there are many interfaces with IUC=1. Especially the interface scatter plots of larger projects such as Vuze hint at a relation between the number of SCC and IUC. Classes with low IUC values tend to have many changes, and there are few classes with high IUC values and many changes.

For concrete classes the correlation coefficients for IUC are less high than those of interfaces, with a median value of -0.32. We identify two causes for these lower values:

- The calculation of IUC for concrete classes only takes into account public methods, while concrete classes contain more methods. In other words, for interfaces IUC is directly related to NOM, where for concrete classes it is related to Number of Public Methods (NOPM).
- As described earlier, concrete classes contain more data than interfaces, and thus have more reasons for change.

Note that the first cause hints at the relation between IUC and NOM, which is investigated in Section 8.3.

Parameter Cohesion

In this investigation we have included two parameter cohesion metrics: Cohesion among Methods in a Class (CAMC) and Normalized Hamming Distance (NHD).

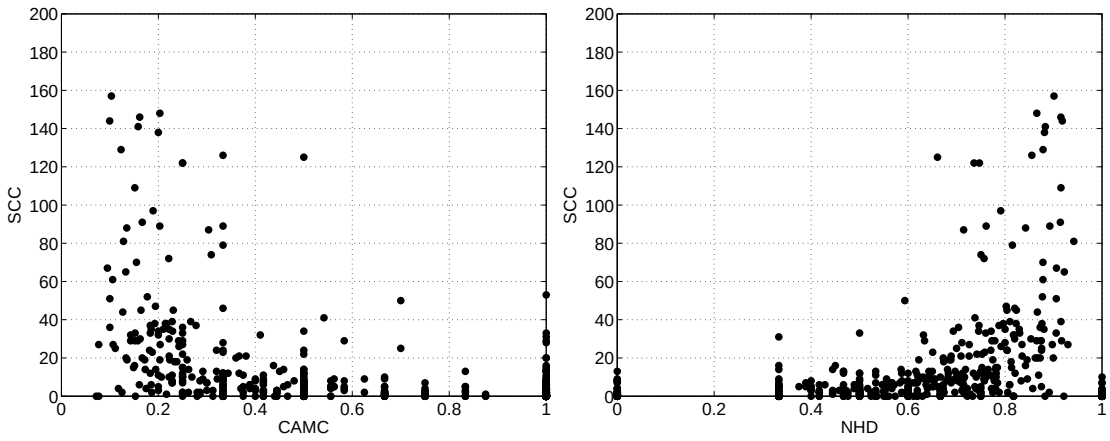
Analysis of CAMC shows that for both interfaces and concrete classes, CAMC often has substantial correlations with SCC. Table 8.4 shows us that CAMC does have substantial correlations, but the median value is not substantial. The Wilcoxon test rejects the hypothesis that the median could be substantial. Interestingly, Table 8.5 shows that the median correlation for the concrete classes for CAMC is 0.52, and the Wilcoxon test accepts the hypothesis $Median(CAMC) = 0.5$ against $Median(CAMC) < 0.5$ at a significance level of 0.05.

Further analysis learns us that there are few classes with a high CAMC and many changes. Furthermore, the metric distribution table (Table D.2) shows that a large percentage of all classes has a CAMC value of 1.

Although the Wilcoxon indicates that CAMC does not have substantial correlations with SCC, the project selection does have an impact on the results. As we have seen earlier in this chapter, the projects ArgoUML-app, JEdit and JFreeChart do not have any substantial correlations for the interfaces. It thus might seem fair to exclude these projects from the analysis. The null-hypothesis that the correlation can be substantial can be accepted based on the outcome of the Wilcoxon test on the reduced dataset.

NHD has a median of 0.35 for interfaces and 0.37 for concrete classes. As can be expected with a low median value, the Wilcoxon test rejects the hypothesis that the median correlation is substantial.

An interesting difference between CAMC and NHD appears when we compare the scatter plots for the project Vuze for the interfaces, see Figure 8.2. If we invert the x-axis for one of the two scatter plots, the figures seem to look very similar. However,



1: Vuze CAMC (Corr:0.62) 2: Vuze NHD (Corr:0.35)

Figure 8.2: Scatter plots that show the difference between NHD and CAMC for interfaces of Vuze.

the correlation coefficient computed for CAMC is nearly twice as large as the one for NHD. This can be explained by the following causes:

- The NHD metric is undefined for interfaces with only a single method.
- CAMC has a relation with NOM as lower bound, where NHD has values of 0.

8.2.3 Conclusions

In the previous section we have investigated the correlation between cohesion metrics and the number of SCC. Based on the analysis, we are now able to accept or reject the first hypothesis for each cohesion metric:

- IIC: reject H_1.
- IUC: partially accept H_1.
- LCOIC: reject H_1.
- NHD: reject H_1.
- CAMC: partially accept H_1.

The rejection of IIC, LCOIC and NHD can be directly justified by the data. These metrics have few significant correlations with the number of SCC, and even less correlation coefficients are substantial. IUC is partially accepted as the data hints at a substantial correlation between IUC and SCC for interfaces. The fact that IUC correlates less with SCC for concrete classes can have multiple causes, such as the relation with the size of

a class. CAMC is partially accepted as it does have a substantial correlation with SCC for concrete classes, and because it is substantially correlated with SCC for interfaces if we remove the projects that do not correlate with any of the metrics.

For our other experiments, the outcomes have the following implications:

- We should keep in mind that some of the projects are less useful in the statistical analysis.
- The metrics for which H_1 is accepted are the primary focus of the rest of the investigation.

8.3 Correlation analysis between cohesion metrics and size

In the previous section the cohesion metrics IUC and CAMC showed a promising correlation with the number of fine-grained changes. Interestingly, the correlation coefficients often seem to be similar to NOM, hinting at a dependency between these metrics. In Chapter 3 the definitions for the metrics are defined. As described in that chapter, the minimum value for IUC is $min(IUC) = \frac{1}{NOM}$, and a similar minimum value applies to CAMC. Hypothesis 2 investigates the relation between the cohesion metrics and size metrics in more detail:

H_2: The cohesion metrics are correlated with the size of Java classes.

8.3.1 Results

The correlation coefficients between the size and cohesion metrics are calculated. Recall from Chapter 4 that NOM is the size metric for interfaces, and NOM and LOC are the size metrics for concrete classes. The results of this analysis can be found in the following tables:

- Table 8.6: Cohesion metrics and NOM for interfaces.
- Table 8.8: Cohesion metrics and LOC for concrete classes.
- Table 8.7: Cohesion metrics and NOM for concrete classes.

Statistics about the correlations per metric are found in Tables 8.9 and 8.10.

8.3.2 Analysis

General observations

The cohesion metrics seem to be strongly correlated with NOM for interfaces. Most correlations are significant, although differences between some metrics become visible.

Project	IUC	CAMC	NHD	IIC	LCOIC
ArgoUML-app	-0.76**	-0.63**	0.44**	-0.36**	0.52**
Eclipse.debug.ui	-0.78**	-0.70**	0.22	-0.45**	0.38**
Eclipse.jdt.core	-0.74**	-0.79**	0.64**	-0.30**	0.40**
Eclipse.Team.core	-0.90**	-0.78**	0.71**	-0.38*	0.56**
Hibernate3	-0.82**	-0.82**	0.62**	-0.32**	0.46**
Jabref	-0.87**	-0.66**	0.37	-0.48**	0.69**
JEdit	-0.57**	-0.53**	0.22	-0.49**	0.25
Jena2	-0.79**	-0.82**	0.56**	-0.03	0.58**
JFreeChart	-0.83**	-0.70**	-0.15	-0.19	0.35**
Joone	-0.86**	-0.66**	0.78**	-0.03	0.30
Log4J	-1.00**	-0.83**	0.37	-0.05	0.51*
Lucene	-0.87**	-0.64**	0.68**	-0.24	0.42**
RapidMiner	-0.83**	-0.83**	0.66**	-0.47**	0.52**
Sweet3D	-0.89**	-0.78**	0.58	-0.67**	0.60**
TripleA	-0.86**	-0.75**	0.61**	-0.50**	0.57**
Vuze	-0.84**	-0.78**	0.46**	-0.37**	0.58**
Xerces	-0.82**	-0.70**	0.70**	-0.29**	0.44**

Table 8.6: Spearman's rank correlation between cohesion metrics and NOM for interfaces. (* indicates a significant correlation at $\alpha = 0.05$, ** at $\alpha = 0.01$)

Project	IUC	CAMC	NHD	LCOIC	LCOM
ArgoUML-app	-0.71**	-0.80**	0.88**	0.32**	0.49**
Eclipse.debug.ui	-0.60**	-0.83**	0.95**	0.40**	0.76**
Eclipse.jdt.core	-0.50**	-0.79**	0.94**	0.33**	0.66**
Eclipse.team.core	-0.46**	-0.79**	0.91**	0.27*	0.56**
Hibernate3	-0.52**	-0.77**	0.86**	0.17**	0.78**
Jabref	-0.62**	-0.70**	0.93**	0.39**	0.80**
JEdit	-0.63**	-0.80**	0.94**	0.42**	0.69**
Jena2	-0.70**	-0.86**	0.81**	0.23**	0.60**
JFreeChart	-0.58**	-0.84**	0.92**	0.11	0.30**
Joone	-0.70**	-0.85**	0.93**	0.27**	0.50**
Log4J	-0.61**	-0.84**	0.95**	0.26**	0.57**
Lucene	-0.55**	-0.80**	0.82**	0.25**	0.70**
RapidMiner	-0.73**	-0.75**	0.93**	0.39**	0.49**
Sweet3D	-0.74**	-0.87**	0.92**	0.57**	0.74**
TripleA	-0.63**	-0.75**	0.92**	0.34**	0.81**
Vuze	-0.48**	-0.51**	0.39**	0.19**	0.41**
Xerces	-0.73**	-0.79**	0.90**	0.28**	0.60**

Table 8.7: Spearman's rank correlation between cohesion metrics and NOM for concrete classes. (* indicates a significant correlation at $\alpha = 0.05$, ** at $\alpha = 0.01$)

Project	IUC	CAMC	NHD	LCOIC	LCOM
ArgoUML-app	-0.58**	-0.71**	0.74**	0.24**	0.49**
Eclipse.debug.ui	-0.41**	-0.80**	0.79**	0.38**	0.70**
Eclipse.jdt.core	-0.45**	-0.72**	0.77**	0.32**	0.53**
Eclipse.Team.core	-0.19	-0.69**	0.69**	0.24	0.46**
Hibernate3	-0.46**	-0.66**	0.68**	0.18**	0.60**
Jabref	-0.36**	-0.55**	0.72**	0.30**	0.72**
JEdit	-0.42**	-0.68**	0.76**	0.36**	0.61**
Jena2	-0.52**	-0.77**	0.69**	0.17**	0.67**
JFreeChart	-0.45**	-0.75**	0.76**	0.08	0.41**
Joone	-0.51**	-0.75**	0.72**	0.24**	0.43**
Log4J	-0.46**	-0.81**	0.83**	0.28**	0.54**
Lucene	-0.41**	-0.71**	0.61**	0.17**	0.67**
RapidMiner	-0.44**	-0.61**	0.69**	0.29**	0.41**
Sweet3D	-0.38**	-0.71**	0.60**	0.34**	0.55**
TripleA	-0.39**	-0.68**	0.77**	0.25**	0.76**
Vuze	-0.51**	-0.62**	0.65**	0.30**	0.68**
Xerces	-0.48**	-0.80**	0.75**	0.19**	0.54**

Table 8.8: Spearman's rank correlation between cohesion metrics and LOC for concrete classes. (* indicates a significant correlation at $\alpha = 0.05$, ** at $\alpha = 0.01$)

Metric	Median	IsSubst	IsStrong
IUC	-0.83	Yes	Yes
CAMC	-0.75	Yes	Yes
NHD	0.58	Possibly	No
LCOIC	0.51	Possibly	No
IIC	-0.36	No	No

Table 8.9: Summary of the correlation between NOM and cohesion metrics for interfaces.

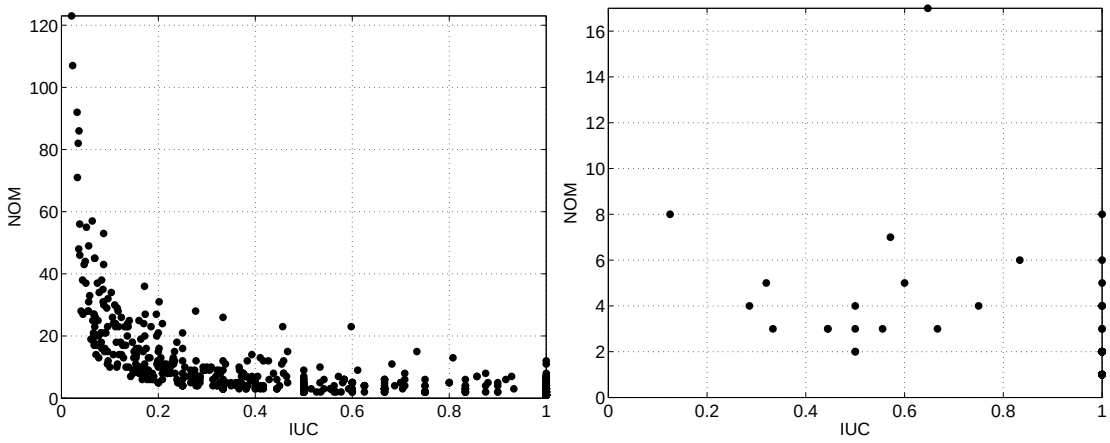
Metric	Median	IsSubst	IsStrong
IUC	-0.62	Yes	No
CAMC	-0.80	Yes	Yes
NHD	-0.92	Yes	Yes
LCOIC	-0.28	No	No
LCOM	0.60	Yes	No

(a) Metrics versus NOM

Metric	Median	IsSubst	IsStrong
IUC	-0.45	No	No
CAMC	-0.71	Yes	Yes
NHD	0.72	Yes	Yes
LCOIC	0.25	No	No
LCOM	0.55	Yes	No

(b) Metrics versus LOC

Table 8.10: Summary of the correlation between size and cohesion metrics for concrete classes.



1: Vuze 2: JEdit

Figure 8.3: Scatter plots that show the relation between NOM and IUC for interfaces of two projects.

If we look at the individual projects, recall that ArgoUML-app, JEdit and JFreechart are three projects with low correlations between the cohesion metrics and SCC. As we can see in Table 8.6, JEdit is the project which has weak correlations. The comparison of Figures 8.3.1 and 8.3.2 shows us that JEdit has few unique data points in the scatter plot, which hints at a lack of data.

Implementation cohesion

As can be seen clearly from Tables 8.6 and 8.9, IIC has relatively few significant correlations -even less substantial correlations- with the interface size metric NOM. The Wilcoxon tests for the median reject the hypothesis that the correlation is substantial.

The low correlations between IIC and NOM can be explained by the same reasons IIC correlates weakly with SCC. Many of the interfaces are completely implemented, and thus IIC=1. Other interfaces do not have any implementing classes, and are excluded from the analysis. Smaller datasets are likely to result in less significant correlations.

Usage cohesion

IUC is strongly correlated with NOM for interfaces. Figure 8.3.1 shows the scatter plot for the project Vuze. Especially the minimum value for IUC ($min(IUC) = \frac{1}{NOM}$) is visible in the scatter plot. For concrete classes, the correlation between IUC and the size metrics is less strong. This can be explained by the relation between IUC and NOM, and the calculation method of IUC explained in Chapter 4. IUC is calculated for only the public methods of concrete classes, and thus the relation between IUC and NOM is less strong for concrete classes.

Similarly, LCOIC does have a relation with the NOM for interfaces and NOPM for concrete classes. However, the correlation between LCOIC and NOM seems less strong than the correlation between IUC and NOM.

Based on the results of the Wilcoxon tests, we accept the hypothesis that IUC is substantially correlated with NOM. For LCOIC, we reject H_2.

Parameter cohesion

In line with the results of IUC, CAMC seems to be strongly correlated with the NOM. The Wilcoxon tests reject the hypothesis that the correlation between CAMC and NOM is not strong, for both interfaces and concrete lasses.

Interestingly, Tables 8.9 and 8.10 show us that NHD seems to correlate substantially with NOM for interfaces, but strong for concrete classes. Note that NHD is calculated using all methods, where IUC and LCOIC are calculated for only the public methods. Although the range of NHD is [0-1], NHD and NOM seem to have a similar relation as IUC and NOM, which is illustrated in Figure 8.4.

One possible explanation for the different correlations for concrete classes and interfaces lies in the dataset size. In Table 8.6 we can see that NHD has relatively high correlations for RapidMiner, Vuze and Hibernate3. For the smaller projects Log4J and JFreeChart the correlation is less strong, and not even significant.

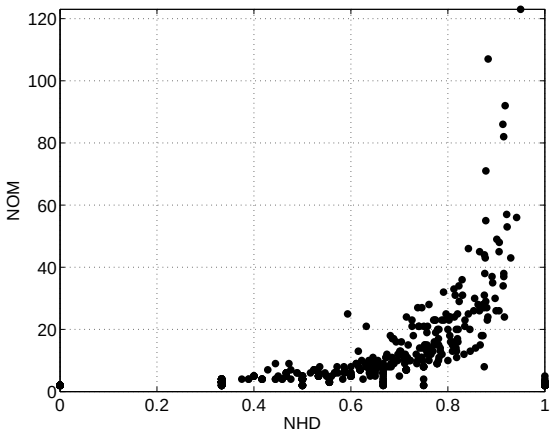


Figure 8.4: Scatter plot that shows the relation between NOM and NHD for interfaces of Vuze.

8.3.3 Conclusions

Based on the results of the correlation analyses, we draw the following conclusions for Hypothesis 2:

- IUC: accept H_2 for interfaces and concrete classes.

- CAMC: accept H_2 for interfaces and concrete classes.
- NHD: partially accept H_2 for interfaces, accept for concrete classes.
- LCOIC: partially accept H_2 for interfaces, reject for concrete classes.
- IIC: reject H_2 for interfaces.

8.4 Summary of the results

In this chapter we have performed the correlation analyses described in Chapter 5.

First we investigated the correlation between the cohesion metrics and the number of SCC. Based on the results of the analysis, we partially accept H_1 for IUC and CAMC For the other metrics, the hypothesis is rejected.

The second investigation is the correlation analysis between the cohesion metrics and size metrics. Although the results are different for interfaces and concrete classes, we accept H_2 for IUC and CAMC. The metrics LCOIC and NHD also show substantial correlations with the class size metrics, and thus the hypothesis is partially accepted.

Concluding, the correlation analyses show promising results for the cohesion metrics IUC and CAMC. They are correlated with the #SCC, and the supporting plots show a clear relation between the two. However, the reason for these correlation might be the fact that the cohesion metrics are correlated with the size of classes. The acceptance of H_2 for the two metrics seems to support this theory. The next chapter investigates whether the cohesion metrics can improve change prediction models based on size.

Chapter 9

Results of the Prediction Models

Based on the correlation analyses, we construct change prediction models to investigate whether cohesion metrics can improve prediction models based on size metrics. In this chapter, we try to find evidence to accept or reject the third hypothesis:

> H_3: The cohesion metrics can improve the performance of prediction models to classify Java classes into change- and not change-prone.

9.1 Overview

Table 9.1 shows the input models we compare in this investigation, as described in Chapter 6. Note that IUC and CAMC are selected as promising metrics, based on the partial acceptance of H_1 for these metrics.

Model	Input metrics
Baseline	Size metrics.
All	All size and cohesion metrics.
Promising	Size metrics, IUC and CAMC.

Table 9.1: Baseline and extended prediction model

Using these input models, we perform the three experiments described in Chapter 6. Each of the experiments has a different goal, as can be seen in Table 9.2.

Section 9.2 and Section 9.3 describe the core experiments for respectively interfaces and concrete classes. In Section 9.4 the experiments with different cutoff points for the definition of change-proneness are described. The third experiment is described in Section 9.5, which investigates whether cohesion metrics of an early release can predict 'future' changes. We report on the results of the experiments in the same structured way as with the correlation analyses: Results, Analysis and Conclusions.

Finally, Section 9.6 gives a short summary of the results of this chapter.

Experiment	Goal
Core experiment	Base experiment to investigate the hypothesis.
Different change-prone cutoff points	Do different cutoff points for the change-prone definition lead to better prediction models?
Predict between releases	Investigate H_3 between releases of a software project.

Table 9.2: The selection of experiments.

9.2 Core experiments

This chapter investigates if cohesion metrics can improve the performance of classification models that are able to classify interfaces as change-prone or not change-prone. The core experiments investigate this goal, using the following characteristics:

- A class is considered change-prone if it has more changes than the median number of changes of all classes in that project.
- The source code metrics are computed on the latest version of the project.
- All available significant changes in the versioning history are included.

Based on the results of the correlation analysis and sizes of the datasets, we have performed the core experiments for the following projects:

Project	Size	MedianSCC
Combined*	194	1
Eclipse**	129	1
Hibernate3	106	1
Jena2	111	4
Vuze	517	4

Table 9.3: Selected projects for core experiments (*Combination of ArgoUML-app, Jabref, Joone, Log4J, Sweet Home 3D, TripleA and Xerces, ** Combination of Eclipse.debug.ui, Eclipse.jdt.core and Eclipse.team.core).

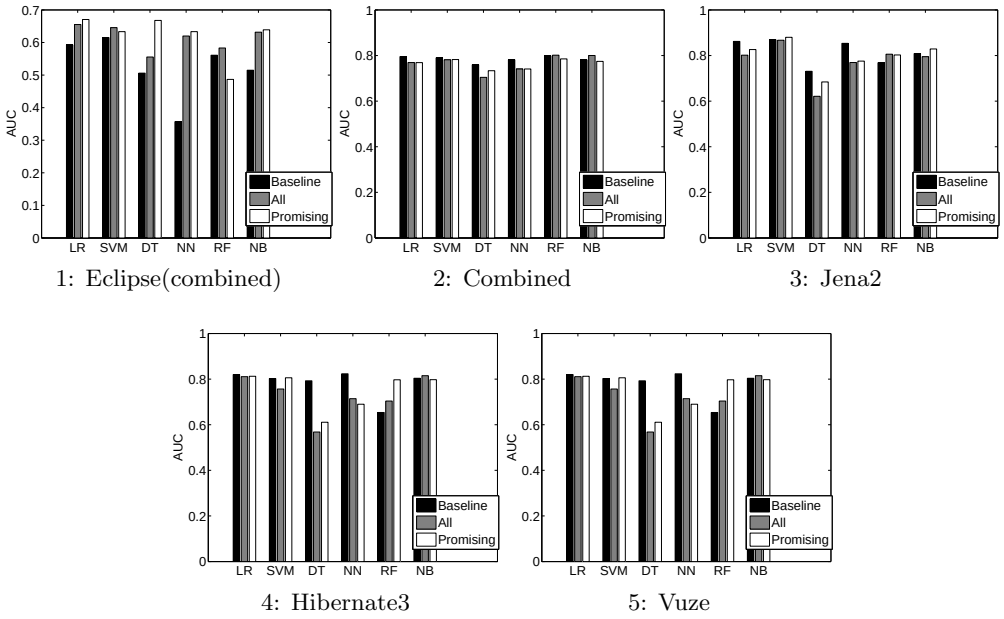
Note that we have combined the datasets of several projects in order to be able to analyze them. We have excluded some of the larger projects with few interface changes, such as RapidMiner, JEdit and JFreeChart.

9.2.1 Results

Figure 9.1 shows the area under the curve (AUC) of the Receiver Operating Characteristic (ROC) curves for this experiment. The abbreviations on the X-axis stand for the

following classification algorithms: Logistic Regression (LR), Support Vector Machine (SVM), Decision Tree (DT), Neural Network (NN), Random Forest (RF) and Naive Bayes (NB).

For each classification algorithm, the graphs show three bars. The most left bar is the performance of the baseline model, the middle bar is the 'all model' and the right bar is the 'promising model'. If we find that one of the extended models performs significantly better than the baseline model, we have evidence that supports H_3.



1: Eclipse(combined) 2: Combined 3: Jena2

4: Hibernate3 5: Vuze

Figure 9.1: AUC values for the prediction models of the core experiments. (Classes: interfaces; Source code model: latest version; SCC: All; Change-prone cutoff point: median)

9.2.2 Analysis

The combined Eclipse projects show low AUC values for all prediction models. In some cases, the AUC value is actually less than 0.5, which indicates that the prediction model performs worse than a random classifier. As described in Chapter 6, negative predictors can be inverted and thus we can interpret the value as $1 - AUC$.

An interesting observation is that the alternative models seem to have higher AUC values compared to the baseline models for the combined Eclipse projects. This might indicate that the cohesion metrics are indeed able to slightly improve the prediction model for this specific project. The figures in Appendix E.1 of the other performance metrics F-Measure and Recall support this theory. However, the improvement can be

explained by the fact that the baseline models for the Eclipse projects provide predictors that have hardly more predictive power then a random guess.

Vuze is by far the largest project in this experiment, and thus might be one of the most interesting projects. The AUC values show an improvement for the Random Forest algorithm, while the other algorithms perform similar or worse with the additional input metrics. Interestingly, the Matthews correlation coefficient is higher for the baseline model than the other two models for the RF algorithm.

The other three projects show AUC values around 0.8 for most projects. The Random Forest algorithm is the only algorithm that sometimes seems to benefit from the additional cohesion metrics as input. The AUC, F-Measure and recall seem to support this theory, while MCC and precision do not.

9.2.3 Conclusions

The performances of the different algorithms and models show that in some specific cases, the cohesion metrics seem to improve the prediction models. However, when analyzed in more detail, the improvements on the baseline model seem too small to draw any solid conclusions. The random forest algorithm seems to produce better models with the cohesion metrics in some cases, but does not perform significantly better than other algorithms without cohesion metrics. Concluding, the core experiments provide no solid evidence that supports our hypothesis.

9.3 Concrete classes

The projects selected for this experiment are shown in Table 9.4. Observe that the median SCC is higher for concrete classes than we have seen in the previous section for interfaces. Compared to the experiments with interfaces, there are more classes and there is more change data available for the experiment with concrete classes.

9.3.1 Results

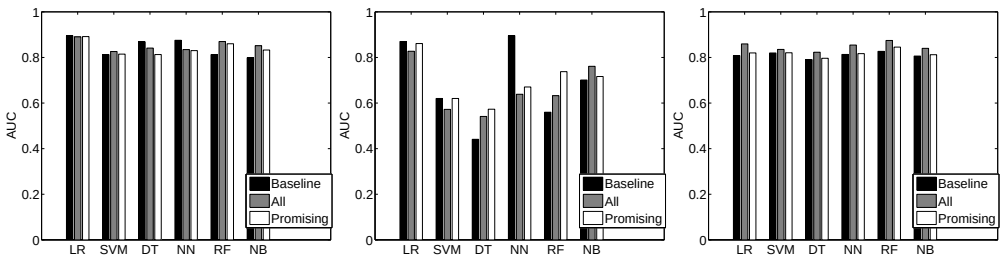
All bar plots of the concrete prediction models can be found in Appendix E.2. As the results for most projects are similar, we present the AUC plots for three projects in Figure 9.2.

9.3.2 Analysis

The results of the concrete classes show no convincing evidence to support the theory that the interface cohesion metrics improve the performance of the selected change prediction models. Figure 9.2 shows three of the AUC values for concrete classes. The Eclipse.jdt.core project shows similar AUC values as the project Vuze. The extended models seem to improve AUC for the prediction models, but only by a very small value. Interesting is the difference of the performance of the Eclipse projects for interfaces,

Project	Size	MedianSCC
ArgoUML-app	615	5
Eclipse.debug.ui	288	7
Eclipse.jdt.core	737	9
Hibernate3	724	7
JabRef	331	5
JEdit	235	7
Jena2	431	7
JFreeChart	280	10
Joone	155	9
Log4J	100	8
Lucene	214	9
RapidMiner	1255	5
Sweet Home 3D	126	1
TripleA	434	7
Vuze	1268	26
Xerces	314	10

Table 9.4: Selected projects for core experiments (concrete classes).



1: Eclipse.jdt.core 2: Sweet Home 3D 3: Vuze

Figure 9.2: AUC values for the prediction models of three projects. (Classes: concerete classes; Source code model: latest version; SCC: All; Change-prone cutoff point: Median)

and the Eclipse projects for concrete classes. We think there are three causes for the differences:

- The combination of three projects leads to bad prediction algorithms. As described by Zimmermann, cross-project prediction models often does not show good results [47].
- The dataset of the combined Eclipse interfaces is still too small to train and test proper prediction models.
- The interfaces change less often than the concrete classes. Chapter 7 shows for Eclipse.jdt.core an average of 15 changes for interfaces, 300 for concrete classes.

Sweet Home 3D is an example of a project where the prediction models perform badly if we look at AUC and MCC. This can be explained by the fact that Sweet Home 3D has few changes available.

In general, the Random Forest algorithm seems to benefit from the cohesion metrics if we look at the different performance metrics. The AUC value for an extended model produced by RF is higher than the AUC value all baseline models of the other classification algorithms for 7/16 projects. However, the differences are never larger than 0.05.

9.3.3 Conclusions

The conclusions are similar to the findings for the experiments for the interfaces. Although some models seem to benefit from the cohesion metrics, the differences in the performance metrics are small. We cannot conclude that the cohesion metrics improve the prediction models.

9.4 Different definitions of change-proneness

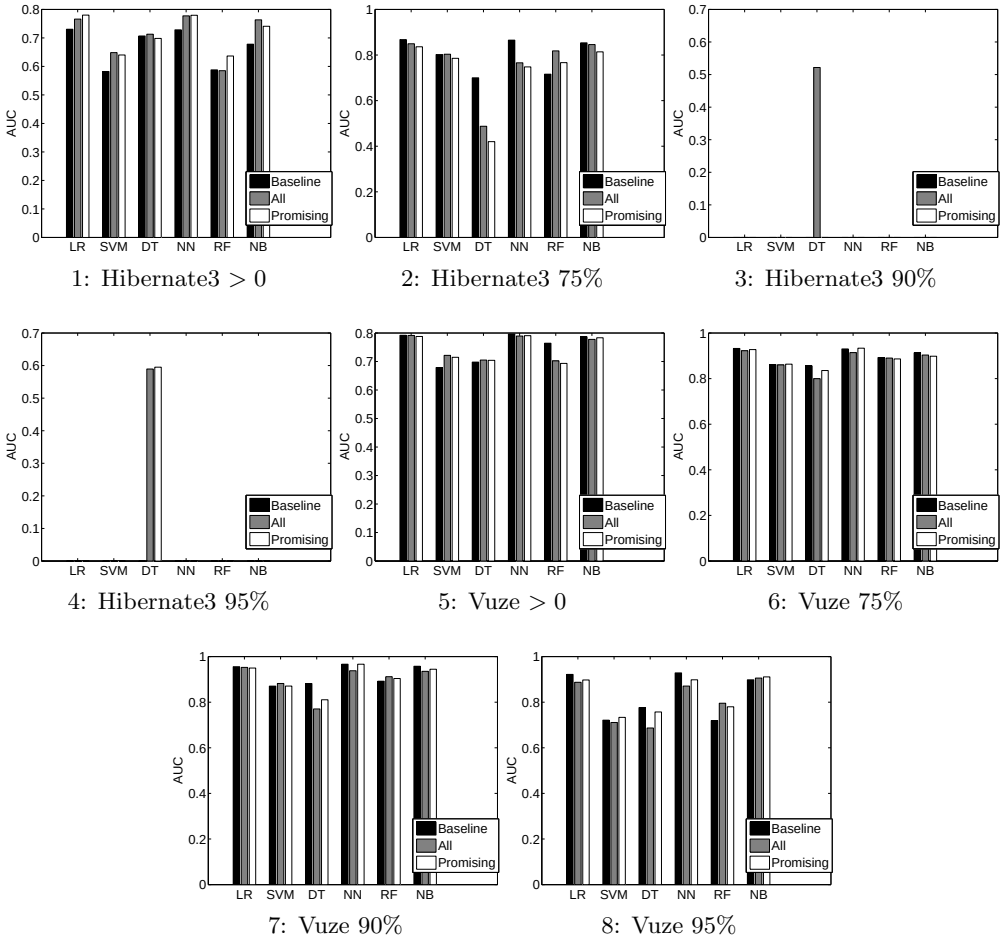
The goal of this experiment is to investigate whether other definitions of change-proneness can produce better prediction models. We have selected four different cutoff points for this analysis: Top 25%, top 10%, top 5% and > 0. We have performed this analysis for two projects, which can be seen in Table 9.5.

Project	Size	Median	> 0	75%	90%	95%
Vuze	517	4	0	12	34	71
Hibernate3	106	2	0	7	24	34

Table 9.5: Selected projects and cutoff points for change-proneness definition.

9.4.1 Results

Figure 9.3 shows the AUC values for the selected models. The supporting metrics can be found in Appendix E.4.



1: Hibernate3 > 0 2: Hibernate3 75% 3: Hibernate3 90%

4: Hibernate3 95% 5: Vuze > 0 6: Vuze 75%

7: Vuze 90% 8: Vuze 95%

Figure 9.3: AUC values for different cutoff points. (Classes: interfaces; Source code model: latest version; SCC: All)

9.4.2 Analysis

As we can see for the project Hibernate3, most of the prediction models for the top 10% and top 5% do not even have an AUC value. This can be explained by the size of the datasets. The top 5% of the dataset is so small that the prediction models do not work properly.

It is interesting to see that the 75% models perform much better than the > 0 models. This can easily be explained by the fact that even small non-cohesive classes do have changes, even though we expect them to change less often than other classes. The > 0 models for Hibernate3 seems to benefit from the cohesion metrics, but the differences are small.

9.4.3 Conclusions

We have found no evidence that the cohesion metrics can improve the performance of change prediction models. Using different change-prone cutoff points does influence the prediction models, but do not show evidence in favor of accepting H_3.

9.5 Predicting between releases

As described in Chapter 6, all of our metrics are calculated using the latest version of the source code of a project. The prediction models are then trained and tested with changes that have happened before that version. This experiment takes into account different releases of the projects, by calculating the cohesion metrics on an early release of the software. The prediction models are then trained and tested with all changes that have occurred after that release.

During the execution of this experiment, we found that the datasets of the projects become much smaller in earlier releases. For many projects, this means we cannot effectively train and test the prediction models, as we do not have enough data to work with.

Since some metrics are undefined for some classes, this is another limitation on the size of the dataset. To cope with this problem, we have used different input models for this analysis. Both the 'All' and the 'Promising' models now only include IUC and NOM.

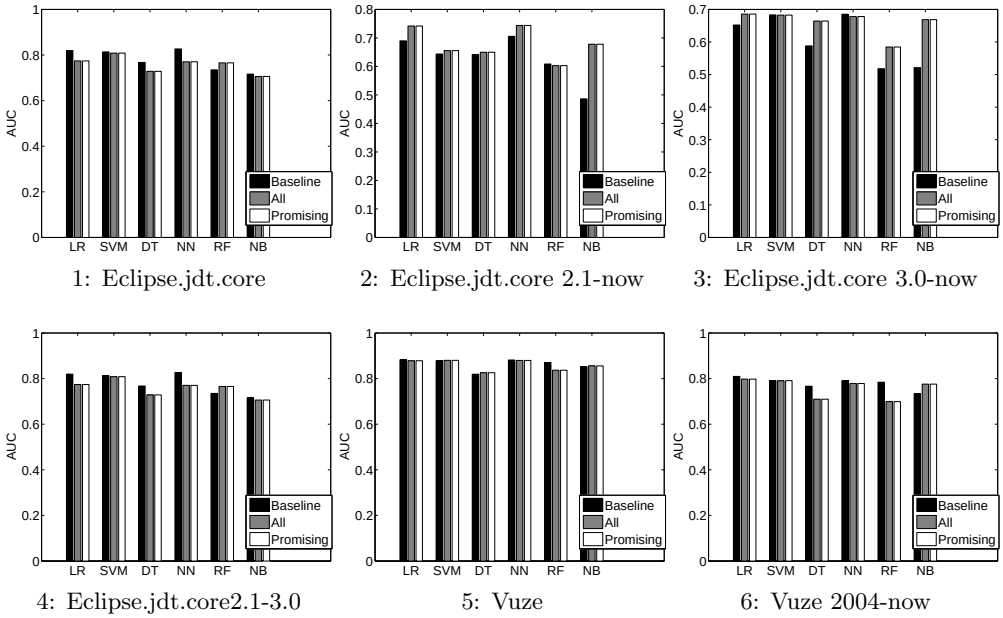
We have performed the analysis for Eclipse.jdt.core and Vuze. For Eclipse.jdt.core, we have selected the 2.1 and 3.0 releases. We have extracted a snapshot of Vuze on the first of January 2004.

Project	Version	Changes	Size	MedianSCC
Eclipse.jdt.core	Latest	All	142	1
Eclipse.jdt.core 2.1-3.0	2.1	2.1-3.0	118	0
Eclipse.jdt.core 2.1-now	2.1	All since 2.1	118	0
Eclipse.jdt.core-now	3.0	All since 3.0	117	0
Vuze	Latest	All	935	1
Vuze 2004-now	2004	All since 2004	112	1

Table 9.6: Selected projects and releases for the prediction models

9.5.1 Results

Table 9.4 shows the AUC values of all models in this experiment. As the results show no clear improvements, the other metrics are found in Appendix E.3.



1: Eclipse.jdt.core 2: Eclipse.jdt.core 2.1-now 3: Eclipse.jdt.core 3.0-now

4: Eclipse.jdt.core2.1-3.0 5: Vuze 6: Vuze 2004-now

Figure 9.4: AUC values for prediction models on different releases. (Classes: interfaces; Source code model: latest version; SCC: between releases; Change-prone cutoff point: median)

9.5.2 Analysis

A quick look at the AUC values for the different models shows us similar results as in the previous experiments. Only for Eclipse.jdt.core 2.1-now and Eclipse.jdt.core 3.0-now show that the NB algorithm performs significantly better with IUC. However, the AUC value is not higher than the AUC values for the baseline models using different algorithms.

An interesting result is that the prediction models for Eclipse.jdt.core 2.1-3.0 perform better than the prediction models for 2.1-now. The AUC values are significantly higher, which can be explained by the fact that there are more interfaces with 0 changes, as the change history is shorter.

9.5.3 Conclusions

Although it was quite difficult to get large enough datasets, we managed to find projects that were suitable for this experiment. We have not found any evidence that the prediction models perform better when IUC is added as an input variable.

9.6 Summary of the results

In this chapter we have performed four different experiments to investigate if cohesion metrics can improve the performance of prediction models that are able to classify Java interfaces and classes as change-prone or not change-prone. We have applied several classification algorithms and compared several performance metrics to perform a thorough investigation of the hypothesis.

The first experiment investigates prediction models for Java interfaces using the latest source code and the complete history as input. Although we have found that some algorithms produce slightly better predictors when the cohesion metrics are used as input, the results are too small and can hardly be generalized. Thus, the first experiment provides no solid evidence that cohesion metrics can improve predictors based on size metrics. Similarly, the results of the core experiments for concrete classes show no solid evidence that cohesion metrics can improve these prediction models.

Our second experiment investigates the effects of using different cutoff points for the definition of change-prone. Although the experiment is only executed on one dataset that is large enough, the results do not show that the prediction models benefit from the use of cohesion metrics. Finally, taking into account different releases of the projects do not show any significant improvements on the baseline model.

Based on the results of all the experiments performed, we have found no solid evidence that supports H_3. Therefore, we reject this hypothesis.

Part IV

Conclusions

Chapter 10

Discussion of the Results

The previous chapters have presented the results of the empirical study performed in this thesis. These results have led to the acceptance and rejection of hypotheses defined in Chapter 1. In this chapter, we discuss the implications of these results in more detail. During the research, we have encountered several potential threats to the validity of the results. Each of these threats will be discussed in Section 10.2.

10.1 Implications of the results

In the previous chapters several results have been presented, and their impact on the hypotheses have been discussed. The question we would like to answer is now: What do these results mean?

We started this thesis research with a baseline model which is illustrated in Model 0 in Figure 10.1. It illustrates the idea that a portion of all changes in interfaces are related to size. This model is based on research performed in the past, including [13, 33, 44, 45].

As described in Chapter 2, there are several publications that show a relation between source code metrics and the change- and/or fault-proneness of classes [25, 26]. Other work shows that this relation can often be explained by the confounding effect of size [13, 33, 44, 45].

The research by Romano and Pinzger [36] shows a promising relation between the Interface Usage Cohesion (IUC) metric and the change-proneness of Java interfaces. This research investigates this relation extensively, and analyzes whether the relation might be caused by the confounding effect of size as well.

The first hypothesis investigates whether cohesion metrics are correlated with the number of fine-grained Source Code Changes (SCC). We have partially accepted H_1 for IUC and Cohesion among Methods in a Class (CAMC). Based on this partial acceptation we have constructed different scenarios of the effect of cohesion on change-proneness, as depicted by the Models 1-3.

Hypothesis 2 investigates the correlation between size and the cohesion metrics. The goal is to investigate whether Model 1 can be accepted, or that maybe Model 2 or Model 3 are more accurate representations of the relation between cohesion and change-proneness.

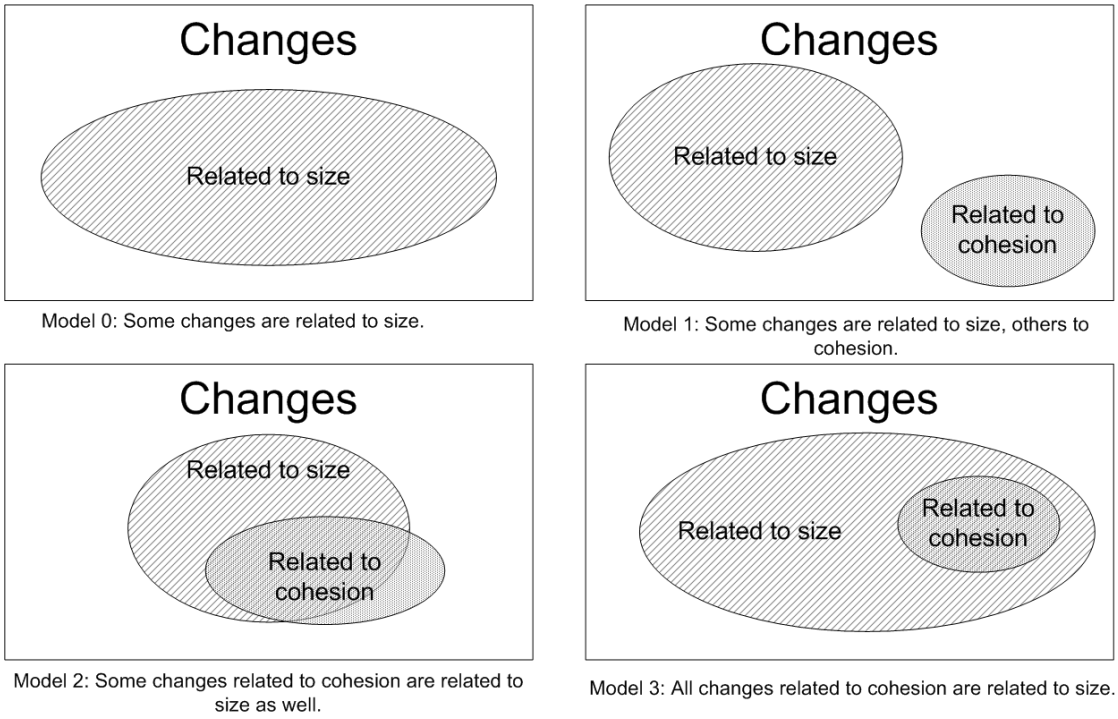


Figure 10.1: Four different change models

The investigation of H_2 is particularly interesting for IUC and CAMC, as H_1 is rejected for the other metrics. As shown in Chapter 8, we have accepted H_2 for both IUC and CAMC. Based on these results, we reject Model 1.

The final hypothesis is constructed to find a definitive answer to the question which model is most accurate. It investigates whether cohesion metrics can improve change prediction models based on size metrics. To investigate the hypothesis, we have designed several experiments to eliminate as many underlying assumptions as possible. Based on the results of the prediction models described in Chapter 9, we have rejected H_3. Apparently, there are not enough changes related to cohesion that are not related to size to significantly improve the prediction models. This implies that there are either very few, or maybe even none of these changes.

Concluding, we have rejected Model 1 based on the (strong) correlation between the cohesion metrics (IUC and CAMC) and the interface size metrics. Based on the results of the prediction models, we did not find evidence for the existence of cohesion related changes that are not size related. Model 3 thus seems the most accurate model.

These results confirm the findings of Zhou [45] on CAMC, and show that IUC behaves similarly. Metrics that seem to be good change predictors, often are unable to improve prediction models based on size metrics. This emphasizes that investigating the confounding effect of class size is an essential aspect of researching change and fault

prediction models.

10.2 Threats to Validity

There are several threats to the validity of the results of this research. Formally, the following threats to validity are discussed in this section:

- Construct validity: Do the operational definitions properly reflect the theoretical meaning of the concepts?
- Internal validity: Are there any confounding variables threatening the results?
- External validity: Can we generalize the findings of the empirical study?
- Statistical validity: Are the correct statistical methods chosen, and are their results interpreted correctly?

Each of the sections below describes a category of threats.

10.2.1 Construct validity

There are several aspects that threaten the construct validity of this research. The threats to construct validity are depicted in Figures 10.2 and 10.3. Figure 10.2 is the conceptual framework presented in the introduction, but now introduces two threats. Similarly, Figure 10.3 illustrates the threats to the operational framework.

(C1) Quality measurement

The first threat challenges the motivation behind this research. The goal of this research is to investigate the relation between cohesion and the change-proneness of classes. The motivation for this goal contains two assumptions. First, we assume that more maintenance is a bad thing. Secondly, we assume that the number of changes is a good reflection of the amount of maintenance performed.

The first assumption is justified in Chapter 1, where we describe that software maintenance costs can account for 40% to 80% of software development costs [19]. The second assumption is more difficult to counter, as each software developer knows that sometimes smaller changes can take up most of the time. This assumption could be mitigated by taking into account the maintenance effort, which is often reported in bug reports. Earlier work by Weiss et al. [41] shows how this can be done.

(C2) Cohesion measurement

In Chapter 3, we have discussed several cohesion metrics. It is a valid question whether the chosen metrics are a good representation of cohesion for interfaces.

The selected cohesion metrics are evaluated in previous research, and the limitations of these metrics are discussed [8, 5, 7, 22]. To our knowledge, Interface Implementation

Cohesion (IIC), IUC and Lack of Cohesion of Interface Clients (LCOIC) have not been evaluated.

(C3) Model selection

For the correlation analyses, we calculate the source code metrics on the latest version of the projects. Then, we calculate the correlation between those metrics and all changes that have led to that version of the source code. A valid question is whether the latest source code model is a good representation of a project. It is not unlikely that a project in an early development phase has a different structure than a project that has been 'stable' for several years.

We deal with this threat in two ways. First of all we include several projects from different sources and in different development phases. Furthermore, in Chapter 6 we describe an experiment that trains prediction models between different releases of a software project.

(C4) Model extraction

The Evolizer framework is able to extract a FAMIX model from the source code. The Evolizer framework has been used in various publications [14, 15, 16, 18, 36], and we have no reason to assume a significant threat.

(C5) Availability

Chapter 7 described the difference between the actual complete versioning history and the versioning history that is in the repository. This difference can have several consequences:

- Limited size of the datasets.
- Some classes might not change anymore, as they have stabilized in an earlier phase.

During the research, we have identified several of these projects, and even excluded some from the empirical study. Others are included, and the results for these projects show low correlations and bad performance of the prediction models.

(C6) Importer

The Evolizer framework that is used to import versioning histories and extract changes is described in Chapter 4. For the importers of the versioning history, we have no reason to assume that they do not import the complete history.

For the Change Distiller, we identified three limitations:

1. If source code files are moved, it does not compare the file at the old location with the file at the new location. Instead, it considers the two files as two different entities.

2. With some complex changes, Change Distiller fails to recognize a modification operation properly. Instead, it recognizes an entity removed change, and an entity addition change.

3. The tree differencing algorithm performs badly on some very complex source code files, due to enormous ASTs.

The first limitation poses a threat to the usability of the data, as we have seen and discussed in several chapters. For each of the limitations, we have no reason to assume that they create a bias in the data.

(C7) Change selection

When we calculate the number of SCC, two decisions are made that could influence the results:

1. We calculate fine-grained SCC, and select only the significant changes.

2. The time interval between which the changes are calculated.

Code churn (i.e. Lines Modified (LM)) could be used as an alternative change metric. Giger et al. [18] show that SCC outperforms code churn in learning bug prediction models.

As described earlier, we calculate the correlation between the metrics on the latest source code, and all available changes of a project. We have trained prediction models between different releases of software to mitigate this threat.

10.2.2 Internal validity

The threats for internal validity concern factors that threaten the independent variables. The relation between size metrics and IUC is a good example of such a threat, which is the reason we have thoroughly studied the relation between the metrics. Furthermore, the datasets are fixed and the metrics are calculated using deterministic algorithms. This ensures that the results can be reproduced.

10.2.3 External validity

(E1) Project selection

The fact that only open source projects have been studied is a threat to *external validity*. In previous research [36], primarily eclipse framework projects were analyzed. In this study, much effort has been put in the project selection. The goal was to get a dataset that is diverse, and we have included several different projects from different sources and communities. For future work, it is interesting to repeat this study on commercial projects.

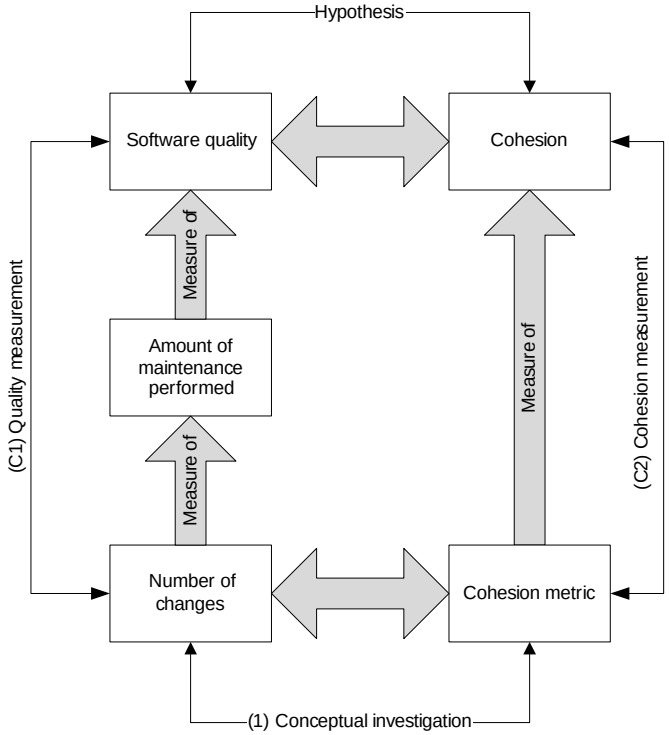


Figure 10.2: Conceptual framework of this research. Each gap represents a threat to the validity of the results.

10.2.4 Statistical validity

Several statistical methods have been applied in this study. For the correlation analysis, Spearman's correlation has been used as it does not make any assumption on the underlying distribution of the data. The statistical significance of the results is described, and *One Sample Wilcoxon Signed-Ranks Tests* are performed to test the median value of the results.

To ensure the validity of the prediction models, we have taken several measures:

- We have included several classification algorithms to avoid a possible bias introduced by one specific algorithm.
- 10 fold Cross-validation is applied.
- Projects with less than 100 data entries have been excluded, as classification models might not produce significant results in those cases.
- We have reported several performance metrics, to avoid reliance on one specific metric.

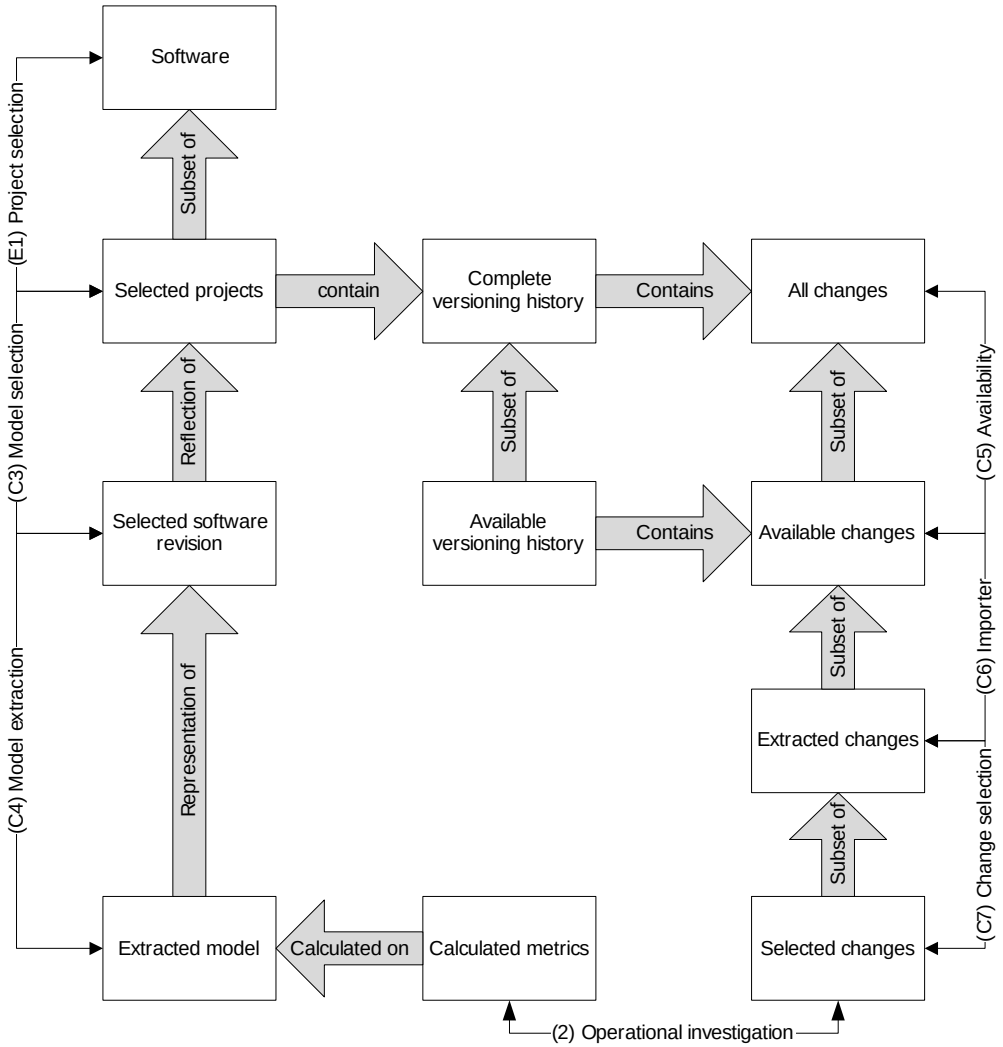


Figure 10.3: Operational framework of this research. Each gap represents a threat to the validity of the results, and each setting represents a threat that we controlled during this research.

Chapter 11

Conclusions and Future Work

11.1 Conclusions

The goal of this thesis is to investigate the impact of cohesion on the change-proneness of Java interfaces. If cohesion has an impact on the change-proneness of interfaces, this information can be used to improve change prediction models based on size metrics. In this thesis, we have performed an empirical study consisting of several experiments to investigate the relation between cohesion and changes.

We have divided the investigation into three hypotheses:

H_1: The cohesion metrics are correlated with the number of fine-grained changes in Java classes.

H_2: The cohesion metrics are correlated with the size of Java classes.

H_3: The cohesion metrics can improve the performance of prediction models to classify Java classes into change- and not change-prone.

The first hypothesis investigates the correlation between interface cohesion metrics and the number of changes of both interfaces and classes. Based on the results of the correlation analysis, we found that:

- Interface Usage Cohesion (IUC) and Cohesion among Methods in a Class (CAMC) are correlated with the number of Source Code Changes (SCC), leading to a partial acceptation of the first hypothesis for these metrics.
- The first hypothesis is rejected for the other cohesion metrics.

We accepted the first hypothesis partially for the two metrics, because the statistical experiments did not provide conclusive evidence to fully support the hypothesis. The results for IUC confirm similar results by Romano and Pinzger [36].

Previous work shows that the relation between source code metrics and change-proneness can often be explained by the confounding effect of size [13, 33, 44, 45]. The fact

that larger interfaces change more frequently is not surprising, and this could be an explanation for the correlation between the cohesion metrics and the number of changes.

The second hypothesis investigates the correlation between the cohesion metrics and size metrics. The results of this correlation analysis show that IUC and CAMC are indeed correlated with Number of Methods (NOM) for interfaces and concrete classes.

To complete this research we investigate whether cohesion metrics can improve change prediction models based on size metrics. These models are able to classify classes as change-prone or not change-prone, based on selected input metrics. We have performed a thorough investigation using several open source projects, but could not find any evidence to accept the third hypothesis.

Concluding, we have not found significant evidence that cohesion metrics can improve change prediction models based on size. Similar to the results of Zhou [45], the relation between cohesion metrics investigated in this research and change-proneness can be explained by the confounding effect of size.

11.2 Future work

This work is an extension of earlier work by Romano and Pinzger [36], as we include more projects, more metrics and more experiments. Still, there is room for improvement.

We have selected several open source projects from different sources and communities, but we have not yet performed this study on proprietary software. Besides analyzing more projects, it might be valuable to improve the research framework. We found that some of the projects did have sufficient data available, but the data could not be used due to limitations in the research framework.

Other improvements to this research include the investigation of other cohesion metrics, and extending the experiments described in this thesis research. Some of the cohesion metrics used in this thesis have limitations which could be resolved through the development of variations on these metrics. However, we think the cohesion metrics selected in this thesis cover most aspects of interface cohesion and we do not expect different results with adjusted cohesion metrics. The experiments with prediction models between different time periods and releases were performed on a limited dataset, and could be extended with more projects and intervals.

Looking at the results of this study, we think that the chances of finding evidence in favor of the third hypothesis are low. In fact, we think the impact of cohesion on change-proneness can be almost fully measured by size metrics, as we have seen in this study. To actually find improvements for prediction models, a different approach might be required.

We think that a qualitative analysis might be a good starting point for finding actual improvements on change prediction models. For instance, it could be interesting to study small classes that change very often, or large classes that rarely change. Analysis of the evolution of such classes should give us answers to questions as: What happened? Did something go wrong, and why? Then, maybe patterns can be detected and even prediction models can be constructed.

Bibliography

[1] Marwen Abbes, Foutse Khomh, Yann-Gael Gueheneuc, and Giuliano Antoniol. An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension. In *Proceedings of the 2011 15th European Conference on Software Maintenance and Reengineering*, CSMR '11, pages 181–190, 2011.

[2] C. Alexander, S. Ishikawa, and M. Silverstein. *A pattern language: towns, buildings, construction*. Center for Environmental Structure series. Oxford University Press, 1977.

[3] Pierre Baldi, Sren Brunak, Yves Chauvin, Claus A. F. Andersen, and Henrik Nielsen. Assessing the accuracy of prediction algorithms for classification: an overview. *Bioinformatics*, 16, 2000.

[4] Jagdish Bansiya, Letha H. Etzkorn, Carl G. Davis, and Wei Li. A class cohesion metric for object-oriented designs. *JOOP*, 11:47–52, 1999.

[5] Challa Bonja and Eyob Kidanmariam. Metrics for class cohesion and similarity between methods. In *Proceedings of the 44th annual Southeast regional conference*, ACM-SE 44, pages 91–95, 2006.

[6] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *Software Engineering, IEEE Transactions on*, 20(6):476–493, 1994.

[7] S. Counsell, S. Swift, A. Tucker, and E. Mendes. Object-oriented cohesion as a surrogate of software comprehension: an empirical study. In *Proceedings of the 5th IEEE Int Source Code Analysis and Manipulation Workshop*, pages 161–169, 2005.

[8] Jehad Al Dallal and Lionel C. Briand. An object-oriented high-level design-based class cohesion metric. *Information and Software Technology*, 52(12):1346 – 1361, 2010.

[9] M. D'Ambros, A. Bacchelli, and M. Lanza. On the impact of design flaws on software defects. In *Proceedings of the 10th International Conference on Quality Software*, pages 23–31, 2010.

[10] Ignatios Deligiannis, Ioannis Stamelos, Lefteris Angelis, Manos Roumeliotis, and Martin Shepperd. A controlled experiment investigation of an object-oriented design heuristic for maintainability. *Journal of Systems and Software*, 72(2):129 – 143, 2004.

[11] Bart Du Bois, Serge Demeyer, Jan Verelst, Tom Mens, and Marijn Temmerman. Does god class decomposition affect comprehensibility? In *IASTED International Conference on Software Engineering*, 2006.

[12] Johann Eder, Gerti Kappel, and Michael Schrefl. Coupling and cohesion in object-oriented systems. Technical report, University of Klagenfurt, 1994.

[13] K. El Emam, S. Benlarbi, N. Goel, and S. N. Rai. The confounding effect of class size on the validity of object-oriented metrics. *Software Engineering, IEEE Transactions on*, 27(7):630–650, 2001.

[14] B. Fluri, H. C. Gall, and M. Pinzger. Fine-grained analysis of change couplings. In *Proceedings of the 5th IEEE Int Source Code Analysis and Manipulation Workshop*, pages 66–74, 2005.

[15] Beat Fluri, Michael Wuersch, Martin Pinzger, and Harald Gall. Change distilling: Tree differencing for fine-grained source code change extraction. *Software Engineering, IEEE Transactions on*, 33:725–743, November 2007.

[16] H. C. Gall, B. Fluri, and M. Pinzger. Change analysis with evolizer and changedistiller. *Software, IEEE*, 26(1):26–33, 2009.

[17] Erich Gamma, Richard Helm, Ralph E. Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[18] Emanuel Giger, Martin Pinzger, and Harald C. Gall. Comparing fine-grained source code changes and code churn for bug prediction. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, MSR '11, pages 83–92, 2011.

[19] R.L. Glass. Frequently forgotten fundamental facts about software engineering. *Software, IEEE*, 18(3):112 –111, may 2001.

[20] Ahmed E. Hassan. Predicting faults using the complexity of code changes. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 78–88, 2009.

[21] Brian Henderson-Sellers. *Object-oriented metrics: measures of complexity*. Prentice-Hall, Inc., 1996.

[22] M. Hitz and B. Montazeri. Measuring coupling and cohesion in object-oriented systems. In *Proceedings of the International Symposium on Applied Corporate Computing*, 1995.

[23] F. Khomh, M. Di Penta, and Y.-G. Gueheneuc. An exploratory study of the impact of code smells on software change-proneness. In *Proceedings of the 16th Working Conference on Reverse Engineering WCRE '09*, pages 75–84, 2009.

[24] Foutse Khomh, Massimiliano Penta, Y.-G. Gueheneuc, and Giuliano Antoniol. An exploratory study of the impact of antipatterns on class change- and fault-proneness. *Empirical Software Engineering*, 6 august 2011:1–33, 2010. 10.1007/s10664-011-9171-y.

[25] T. M. Khoshgoftaar and R. M. Szabo. Improving code churn predictions during the system test and maintenance phases. In *Proceedings of the 1994 International Conference on Software Maintenance*, pages 58–67, 1994.

[26] Wei Li and Sallie Henry. Object-oriented metrics that predict maintainability. *Journal of Systems and Software*, 23(2):111 – 122, 1993.

[27] Robert Cecil Martin. *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall PTR, 2003.

[28] Thomas J. Mccabe. A complexity measure. In *Proceedings of the 2nd International Conference on Software Engineering*, 1976.

[29] Raimund Moser, Witold Pedrycz, and Giancarlo Succi. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 181–190, 2008.

[30] N. Nagappan, T. Ball, and A. Zeller. Mining metrics to predict component failures. In *Proceedings of the 28th International Conference on Software Engineering*, pages 452–461, 2006.

[31] Nachiappan Nagappan and Thomas Ball. Use of relative code churn measures to predict system defect density. In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, pages 284–292, 2005.

[32] Nachiappan Nagappan, Andreas Zeller, Thomas Zimmermann, Kim Herzig, and Brendan Murphy. Change bursts as defect predictors. In *Proceedings of the 2010 IEEE 21st International Symposium on Software Reliability Engineering*, ISSRE '10, pages 309–318, 2010.

[33] Steffen M. Olbrich, Daniela S. Cruzes, and Dag I. K. Sjoberg. Are all code smells harmful? a study of god classes and brain classes in the evolution of three open source systems. In *Proceedings of the 2010 International Conference on Software Maintenance*, ICSM '10, pages 1–10, 2010.

[34] D. L. Parnas. *On the criteria to be used in decomposing systems into modules*, pages 139–150. Yourdon Press, 1979.

[35] M. Perepletchikov, C. Ryan, and K. Frampton. Cohesion metrics for predicting maintainability of service-oriented software. In *Proceedings of the 7th International Conference on Quality Software*, pages 328–335, 2007.

[36] Daniele Romano and Martin Pinzger. Using source code metrics to predict change-prone java interfaces. In *Proceedings of the 27th International Conference on Software Maintenance*, pages 303–312, 2011.

[37] H.D. Rombach. A controlled experiment on the impact of software structure on maintainability. *Software Engineering, IEEE Transactions on*, SE-13(3):344 – 354, march 1987.

[38] A. Schröter, T. Zimmermann, and A. Zeller. Predicting component failures at design time. In *Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering*, pages 18–27, 2006.

[39] J. Śliwerski, T. Zimmermann, and A. Zeller. When do changes induce fixes? In *Proceedings of the 2005 International Workshop on Mining Software Repositories*, pages 1–5, 2005.

[40] W. Stevens, G. Myers, and L. Constantine. *Structured design*, pages 205–232. Yourdon Press, 1979.

[41] Cathrin Weiss, Rahul Premraj, Thomas Zimmermann, and Andreas Zeller. How long will it take to fix this bug? In *Proceedings of the Fourth International Workshop on Mining Software Repositories*, MSR '07, pages 1–, 2007.

[42] Ian H. Witten and Eibe Frank. *Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations (The Morgan Kaufmann Series in Data Management Systems).* Morgan Kaufmann, 1st edition, 1999.

[43] E. Yourdon and L. L. Constantine. *Structured design. Fundamentals of a discipline of computer program and systems design.* Prentice-Hall, 1979.

[44] Nico Zazworka, Michele A. Shaw, Forrest Shull, and Carolyn Seaman. Investigating the impact of design debt on software quality. In *Proceedings of the 2nd Workshop on Managing Technical Debt*, MTD '11, pages 17–23, 2011.

[45] Yuming Zhou, H. Leung, and Baowen Xu. Examining the potentially confounding effect of class size on the associations between object-oriented metrics and change-proneness. *Software Engineering, IEEE Transactions on*, 35(5):607–623, 2009.

[46] T. Zimmermann, A. Zeller, P. Weißgerber, and S. Diehl. Mining version histories to guide software changes. *Software Engineering, IEEE Transactions on*, 31:429–445, 2005.

[47] Thomas Zimmermann, Nachiappan Nagappan, Harald Gall, Emanuel Giger, and Brendan Murphy. Cross-project defect prediction: a large scale experiment on data vs. domain vs. process. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ESEC/FSE '09, pages 91–100, 2009.

Acronyms

AUC area under the curve.

CAMC Cohesion among Methods in a Class.

DT Decision Tree.

FN False Negative.

FP False Positive.

IIC Interface Implementation Cohesion.

IPC Interface Parameter Cohesion.

IUC Interface Usage Cohesion.

LCOIC Lack of Cohesion of Interface Clients.

LCOM Lack of Cohesion of Methods.

LM Lines Modified.

LOC Lines of Code.

LR Logistic Regression.

MCC Matthews Correlation Coefficient.

NB Naive Bayes.

NHD Normalized Hamming Distance.

NN Neural Network.

NOC Number of Clients.

NOIC Number of Implementing Classes.

NOM Number of Methods.

NOPM Number of Methods.

RF Random Forest.

ROC Receiver Operating Characteristic.

SCC Source Code Changes.

SVM Support Vector Machine.

TN True Negative.

TP True Positive.

WMC Weighted Method per Class.

Appendix

Appendix A

Metric Computation Tables

This appendix contains the computation tables for each cohesion metric. The information presented in the tables is discussed in Chapters 3 and 4.

Metric	IUC
Dataset selection	Interfaces: All interfaces that have at least 1 method, that is invoked by at least one client. Concrete classes: All concrete classes that have at least 1 public non-static method that is invoked by at least one client.
Method selection	Interfaces: All methods that are invoked at least once. Concrete classes: All public nonstatic methods that are invoked at least once.

Metric computation	`for each interface i in interfaces`
	`{`
	`  methods=getMethods(i);`
	`  clients=getUniqueClients(methods);`
	`  total_usage=0;`
	`  for each client c in clients{`
	`  invoked=getUniqueInvocations(c,i);`
	`  total_usage+=invoked.size()/methods.size();`
	`  }`
	`  IUC=total_usage/clients.size();`
	`}`

Table A.1: IUC computation

Metric	CAMC
Dataset selection	Interfaces: All interfaces that have at least 1 method, that is invoked by at least one client. Concrete classes: All concrete classes that have at least 1 public non-static method that is invoked by at least one client.
Method selection	All methods.
Parameter selection	All distinct parameter types. Includes primitive types.

Metric computation	`for each interface i in interfaces`
	`{`
	`  methods=getMethods(i);`
	`  param_types=getDistinctParams(methods);`
	`  param_usage=0;`
	`  for each method m in methods{`
	`  params=getDistinctMethodParams(m);`
	`  param_usage+=params.size()/param_types.size();`
	`  }`
	`  CAMC=param_usage/methods.size();`
	`}`

Table A.2: CAMC computation

Metric	NHD
Dataset selection	Interfaces: All interfaces that have at least 1 method, that is invoked by at least one client. Concrete classes: All concrete classes that have at least 1 public non-static method that is invoked by at least one client.
Method selection	All methods.
Parameter selection	All distinct parameter types. Includes primitive types.

Metric computation	`for each interface i in interfaces`
	`{`
	`  methods=getMethods(i);`
	`  param_types=getDistinctParams(i);`
	`  param_usage=0;`
	`  for each param_type t in param_types{`
	`  params=getDistinctMethodParams(m);`
	`  type_usage=countMethodsThatUse(t,i);`
	`  param_usage+=type_usage*`
	`  (methods.size()-type_usage);`
	`  }`
	`  denominator=param_types.size()*`
	`  methods.size()*(methods.size()-1);`
	`  NHD=(2*param_usage)/denominator;`
	`}`

Table A.3: NHD computation

Metric	IIC
Dataset selection	Interface criteria: • Should have at least 1 method, that is invoked by at least one client. • Should be implemented by at least one concrete class. This metric does not exist for concrete classes.
Method selection	All non-static methods.

Metric computation	`for each interface i in interfaces`
	`{`
	`  methods=getMethods(i);`
	`  implementers=getAllImplementingClasses(i);`
	`  impl_usage=0;`
	`  for each implementer j in implementers{`
	`  nonempty_bodies=0;`
	`  for each method m in methods{`
	`  method_body=findImplementation(m,j)`
	`  if isnotempty(method_body){`
	`  nonempty_bodies ++;`
	`  }`
	`  }`
	`  impl_usage=nonempty_bodies/methods.size();`
	`  }`
	`  IIC=param_usage/methods.size();`
	`}`

Find implementation	The findImplementation method walks the inheritance tree starting at the concrete class until it reaches the interface. If the concrete class does not define the method that is declared in the interface, it is likely defined by one of its super classes.
Definition of empty methods	A method is considered empty if it does not have any attached associations in the Famix model, such as field accesses.

Table A.4: IIC computation

Metric	LCOIC
Dataset selection	Interfaces: All interfaces that have at least 1 method, that is invoked by at least one client. Concrete classes: All concrete classes that have at least 1 public non-static method that is invoked by at least one client.
Method selection	Interfaces: All methods that are invoked at least once. Concrete classes: All public non-static methods that are invoked at least once.

Metric computation	`for each interface i in interfaces`
	`{`
	`  P=0;Q=0;`
	`  clients=getInvokingClasses(i);`
	`  for each pair of clients{`
	`  if shareMethods(Client1,Client2)`
	`  Q++;`
	`  else`
	`  P++;`
	`  }`
	`  LCOIC=(P>Q) ? P-Q : 0;`
	`}`

shareMethods	The shareMethods function returns true if two clients share at least one method..

Table A.5: LCOIC computation

Metric	LCOM
Dataset selection	Interfaces: This metric is undefined for interfaces. Concrete classes: All concrete classes that have at least 1 public non-static method that is invoked by at least one client.
Method selection	Interfaces: All methods that are invoked at least once. Concrete classes: All public non-static methods that are invoked at least once.

Metric computation	`for each class c in classes`
	`{`
	`  P=0;Q=0;`
	`  clients=getMethods(c);`
	`  for each pair of clients{`
	`  if shareAttributes(Client1,Client2)`
	`  Q++;`
	`  else`
	`  P++;`
	`  }`
	`  LCOM=(P>Q) ? P-Q : 0;`
	`}`

shareAttributes	The shareAttributes function returns true if two methods share at least one attribute..

Table A.6: LCOM computation

Appendix B

Change Distiller Changetypes

Changetypes	Significant
ADDING_ATTRIBUTE_MODIFIABILITY	Yes
ADDING_CLASS_DERIVABILITY	Yes
ADDING_METHOD_OVERRIDABILITY	Yes
ADDITIONAL_CLASS	Yes
ADDITIONAL_FUNCTIONALITY	Yes
ADDITIONAL_OBJECT_STATE	Yes
ALTERNATIVE_PART_DELETE	Yes
ALTERNATIVE_PART_INSERT	Yes
ATTRIBUTE_RENAMING	Yes
ATTRIBUTE_TYPE_CHANGE	Yes
CLASS_RENAMING	Yes
COMMENT_DELETE	No
COMMENT_INSERT	No
COMMENT_MOVE	No
COMMENT_UPDATE	No
CONDITION_EXPRESSION_CHANGE	Yes
DECREASING_ACCESSIBILITY_CHANGE	Yes
DOC_DELETE	No
DOC_INSERT	No
DOC_UPDATE	No
INCREASING_ACCESSIBILITY_CHANGE	Yes
METHOD_RENAMING	Yes
PARAMETER_DELETE	Yes
PARAMETER_INSERT	Yes
PARAMETER_ORDERING_CHANGE	Yes
PARAMETER_RENAMING	Yes
PARAMETER_TYPE_CHANGE	Yes
PARENT_CLASS_CHANGE	Yes
PARENT_CLASS_DELETE	Yes
PARENT_CLASS_INSERT	Yes
PARENT_INTERFACE_CHANGE	Yes
PARENT_INTERFACE_DELETE	Yes
PARENT_INTERFACE_INSERT	Yes
REMOVED_CLASS	Yes
REMOVED_FUNCTIONALITY	Yes
REMOVED_OBJECT_STATE	Yes
REMOVING_ATTRIBUTE_MODIFIABILITY	Yes
REMOVING_CLASS_DERIVABILITY	Yes
REMOVING_METHOD_OVERRIDABILITY	Yes
RETURN_TYPE_CHANGE	Yes
RETURN_TYPE_DELETE	Yes
RETURN_TYPE_INSERT	Yes
STATEMENT_DELETE	Yes
STATEMENT_INSERT	Yes
STATEMENT_ORDERING_CHANGE	Yes
STATEMENT_PARENT_CHANGE	Yes
STATEMENT_UPDATE	Yes
UNCLASSIFIED_CHANGE	No

Table B.2: Changetypes recognized by Evolizer change distiller

Appendix C

Configurations of the Classification Algorithms

This appendix describes the configuration of the classification algorithms in RapidMiner, as well as the configuration of the cross-validation operator. All experiments have been performed on RapidMiner version 5.2.001. We report on the parameter values only to allow validation of the results. The meaning of the parameters can be found on the website of RapidMiner.[1]

C.1 Logistic Regression

Parameter	Value
R	1.0E-8
M	-1.0

Table C.1: Parameters for operator W-Logistic: Class for building and using a multinomial logistic regression model with a ridge estimator.

C.2 Support Vector Machine

[1] RapidMiner, http://www.rapidminer.com

Parameter	Value
svm type	C-SVC
kernel type	rbf
gamma	0.0
C	0.0
cache size	80
epsilon	0.0010
shrinking	true
calculate confidences	false
confidence for multiclass	false

Table C.2: Parameters for operator SVMlib: This operators is an SVM Learner based on the Java libsvm, an SVM learner.

C.3 Decision Tree

Parameter	Value
criterion	information_gain
minimal size for split	4
minimal leaf size	2
minimal gain	0.05
maximal depth	20
confidence	0.1
number of prepruning alternatives	3
no pre pruning	false
nu pruning	false

Table C.3: Parameters for operator Decision Tree: Generates decision trees to classify nominal data.

C.4 Neural Network

Parameter	Value
training cycles	500
learning rate	0.3
momentum	0.2
decay	false
shuffle	true
normalize	true
error epsilon	1.0E-5
use local random seed	false

Table C.4: Parameters for operator Neural Net: Learns a neural net from the input data..

C.5 Random Forest

Parameter	Value
I	20.0
K	0.0
S	1.0
depth	
D	false

Table C.5: Parameters for operator W-Random Forest: Class for constructing a forest of random trees.

C.6 Naive Bayes

Parameter	Value
laplace correction	true

Table C.6: Parameters for operator Naive Bayes: Returns classification model using estimated normal distributions.

C.7 X-Validation

Parameter	Value
average performances only	true
leave one out	false
number of validations	10
sampling type	stratified sampling
use local random seed	false
parallelize training	false
parallelize testing	false

Table C.7: Parameters for operator X-Validation: X-Validation encapsulates a cross-validation in order to estimate the performance of a learning operator.

Appendix D

Metric Distributions

This appendix contains tables and plots to support the investigation of the correlation analysis between cohesion metrics and the number of SCC. Section D.2 contains scatter plots for the cohesion metrics for all metrics, and Section D.1 contains tables with information about the distribution tables.

D.1 Metric statistics

Project	Count	Min	Median	Mean	Ones	Median($\neq 1$)
Argouml-app	86	0.03	1	0.78	53	0.35
Eclipse.debug.ui	123	0.09	1	0.72	62	0.40
Eclipse.jdt.core	142	0.03	0.5	0.60	50	0.33
Eclipse.team.core	39	0.08	0.75	0.67	17	0.47
Hibernate3	182	0.03	0.86	0.67	87	0.31
Jabref	32	0.11	1	0.83	22	0.50
JEdit	57	0.13	1	0.85	40	0.50
Jena2	173	0.02	0.67	0.65	74	0.36
JFreeChart	72	0.03	1	0.75	39	0.44
Joone	26	0.05	0.47	0.60	11	0.33
Log4j	20	0.14	1	0.77	13	0.25
Lucene	40	0.04	0.71	0.69	19	0.50
Rapidminer	187	0.04	1	0.71	102	0.33
Sweet Home 3D	25	0.04	1	0.73	15	0.34
TripleA	92	0.07	0.75	0.72	44	0.50
Vuze	935	0.02	0.8	0.67	440	0.36
Xerces	92	0.07	0.5	0.59	26	0.41

Table D.1: IUC metric distribution statistics for interfaces.

Project	Count	Min	Median	Mean	Ones	Median($\neq 1$)
Argouml-app	72	0.10	1	0.87	55	0.42
Eclipse.debug.ui	107	0.11	1	0.79	64	0.50
Eclipse.jdt.core	79	0.15	1	0.73	40	0.50
Eclipse.team.core	29	0.17	0.75	0.72	12	0.50
Hibernate3	160	0.06	0.75	0.70	74	0.50
Jabref	24	0.20	1	0.88	17	0.56
Jedit	48	0.33	1	0.86	32	0.53
Jena2	141	0.06	0.75	0.67	67	0.33
JFreeChart	62	0.07	1	0.80	41	0.50
Joone	23	0.11	0.5	0.64	9	0.33
Log4j	18	0.17	1	0.84	14	0.25
Lucene	34	0.17	1	0.87	27	0.33
Rapidminer	162	0.11	1	0.76	91	0.44
Sweet Home 3D	17	0.09	1	0.76	10	0.50
TripleA	79	0.19	1	0.77	41	0.50
Vuze	753	0.06	1	0.71	386	0.40
Xerces	64	0.10	0.53	0.67	27	0.50

Table D.2: CAMC metric distribution statistics for interfaces

Project	Count	Median	Mean	Zeros	Ones
Argouml-app	47	0.50	0.52	8	9
Eclipse.debug.ui	75	0.60	0.58	5	13
Eclipse.jdt.core	66	0.68	0.64	3	3
Eclipse.team.core	24	0.53	0.47	5	0
Hibernate3	119	0.60	0.58	11	7
Jabref	14	0.50	0.59	0	1
Jedit	36	0.50	0.54	6	9
Jena2	114	0.62	0.61	8	8
JFreeChart	40	0.67	0.68	1	14
Joone	18	0.64	0.61	0	0
Log4j	7	0.52	0.52	2	1
Lucene	25	0.40	0.39	10	2
Rapidminer	103	0.53	0.51	17	6
Sweet Home 3D	9	0.67	0.59	2	1
TripleA	56	0.46	0.45	13	4
Vuze	584	0.62	0.58	68	61
Xerces	102	0.44	0.44	24	3

Table D.3: NHD metric distribution statistics for interfaces.

Project	Count	Min	Median	Mean	Zeros	Ones	Median($\neq 1$)
Argouml-app	77	0	1	0.85	3	44	0.67
Eclipse.debug.ui	103	0	1	0.91	3	75	0.75
Eclipse.jdt.core	129	0.08	1	0.91	0	87	0.76
Eclipse.team.core	29	0.42	1	0.91	0	21	0.71
Hibernate3	170	0	1	0.89	2	107	0.75
Jabref	31	0.42	1	0.91	0	19	0.79
Jedit	54	0	1	0.85	1	32	0.67
Jena2	171	0	1	0.87	3	102	0.73
JFreeChart	67	0	1	0.89	2	47	0.75
Joone	24	0.33	0.99	0.91	0	12	0.92
Log4j	20	0.31	1	0.85	0	12	0.64
Lucene	38	0	1	0.89	2	27	0.67
Rapidminer	166	0	1	0.86	2	95	0.75
Sweet Home 3D	25	0.2	1	0.93	0	19	0.90
TripleA	92	0	1	0.91	1	71	0.63
Vuze	879	0	1	0.86	14	521	0.72
Xerces	91	0.38	1	0.91	0	63	0.68

Table D.4: IIC metric distribution statistics for interfaces

Project	Count	Max	Median	Mean	Zeros	Median($\neq 0$)
Argouml-app	86	8391	0	109.81	74	3
Eclipse.debug.ui	123	221	0	2.33	104	3
Eclipse.jdt.core	142	514	0	10.27	115	3
Eclipse.team.core	39	10	0	0.67	31	2.5
Hibernate3	182	6312	0	71.97	142	10
Jabref	32	21	0	1.16	27	3
Jedit	57	8	0	0.26	50	1
Jena2	173	27536	0	479.31	137	46.5
JFreeChart	72	3	0	0.18	65	2
Joone	26	571	0	22.96	20	1.5
Log4j	20	17	0	1.00	17	2
Lucene	40	165	0	4.33	34	1
Rapidminer	187	33814	0	279.93	154	13
Sweet Home 3D	25	253	0	21.52	20	79
TripleA	92	138	0	4.10	73	3
Vuze	935	21698	0	90.56	721	9
Xerces	92	36	0	0.85	82	3.5

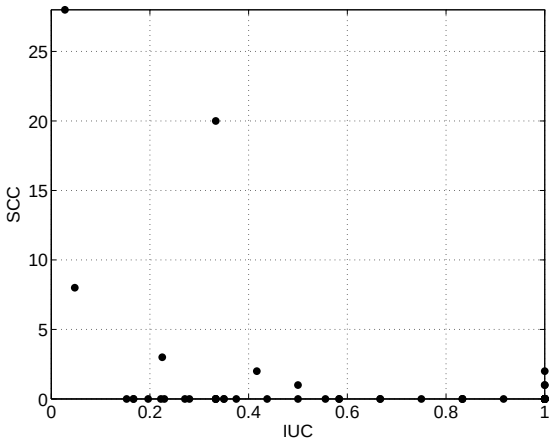
Table D.5: LCOIC metric distribution statistics for interfaces

D.2 Scatter plots

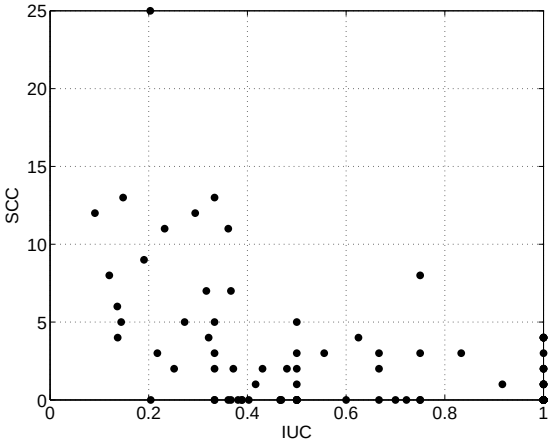
This appendix contains scatter plots of the cohesion metrics versus SCC for the selected projects. They give insight in the distributions of both the cohesion metrics and SCC, and show us the type of the relation between the two. Note that the scatter plots do not give any information about the occurrences of the values. For instance, $(0, 0)$ values or $(0, 1)$ values occur relatively often. That information is found in the tables presented in Section D.1.

- Figure D.1 (page 113): IUC versus SCC for all interfaces.
- Figure D.2 (page 116): CAMC versus SCC for all interfaces.
- Figure D.3 (page 119): Normalized Hamming Distance (NHD) versus SCC for all interfaces.
- Figure D.4 (page 122): Interface Implementation Cohesion (IIC) versus SCC for all interfaces.
- Figure D.5 (page 125): Lack of Cohesion of Interface Clients (LCOIC) versus SCC for all interfaces.
- Figure D.6 (page 129): IUC versus SCC for all concrete classes.
- Figure D.7 (page 132): CAMC versus SCC for all concrete classes.
- Figure D.8 (page 135): NHD versus SCC for all concrete classes.
- Figure D.9 (page 138): LCOIC versus SCC for all concrete classes.

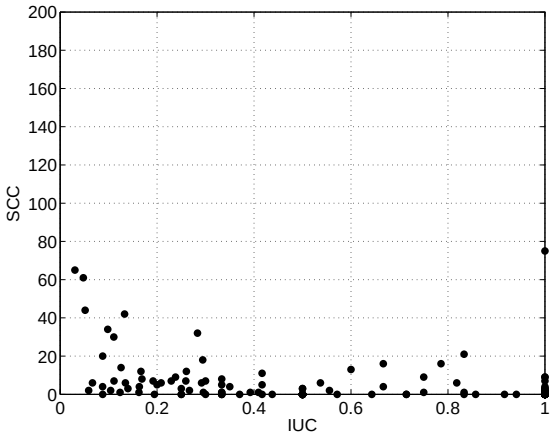
D.2.1 Scatter plots for interfaces



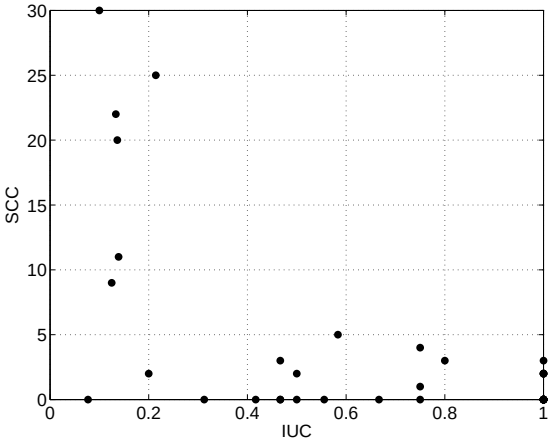
1: ArgoUML-app



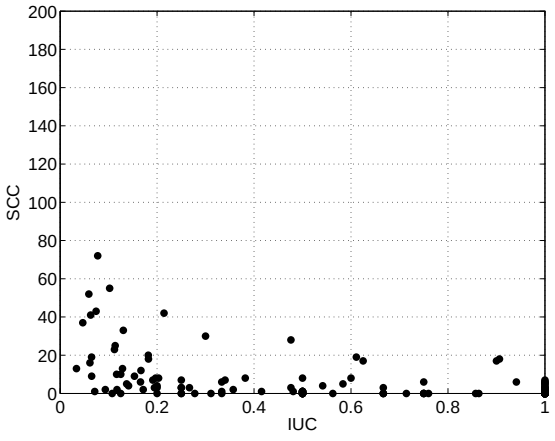
2: Eclipse.debug.ui



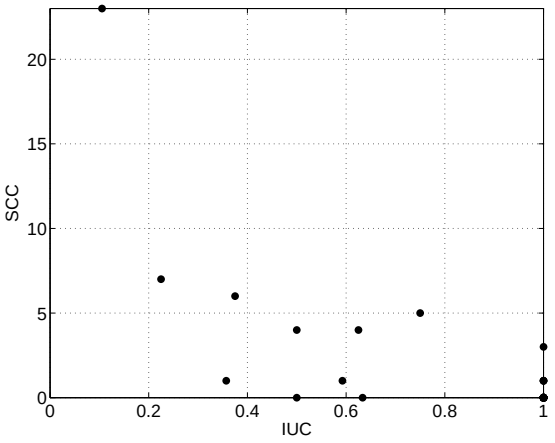
3: Eclipse.jdt.core



4: Eclipse.team.core

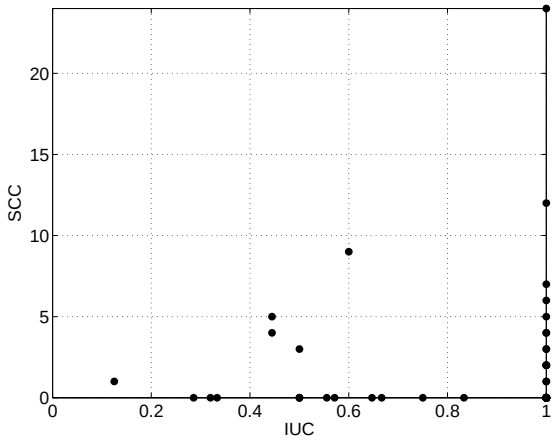


5: Hibernate3

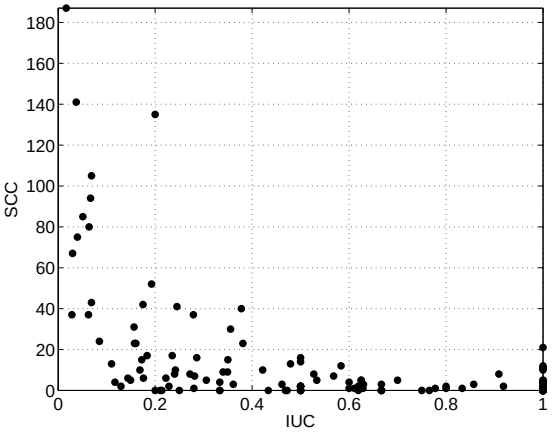


6: Jabref

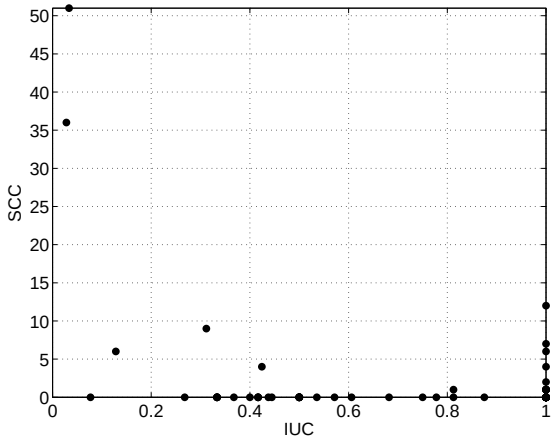
Figure D.1: IUC distribution scatter plots



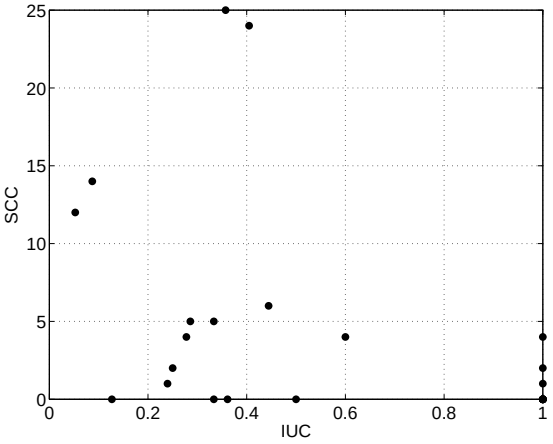
7: Jedit



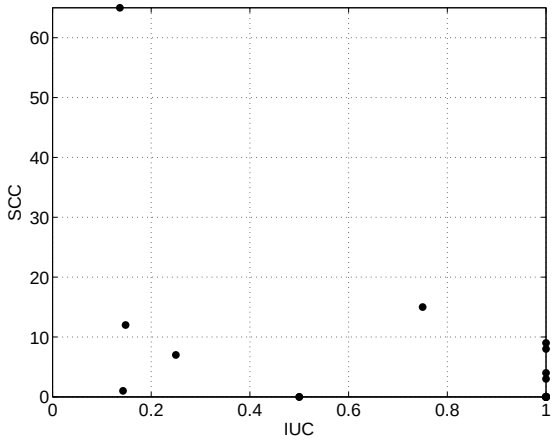
8: Jena2



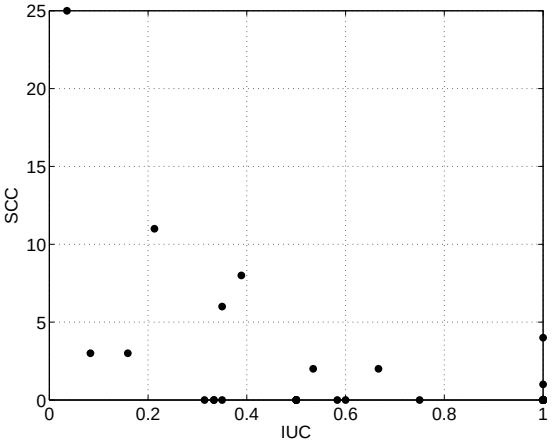
9: JFreeChart



10: Joone

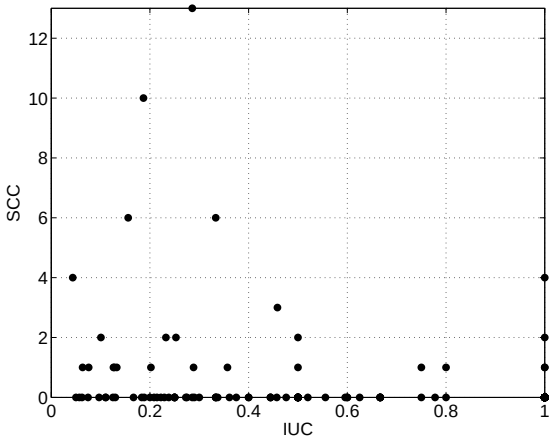


11: Log4j

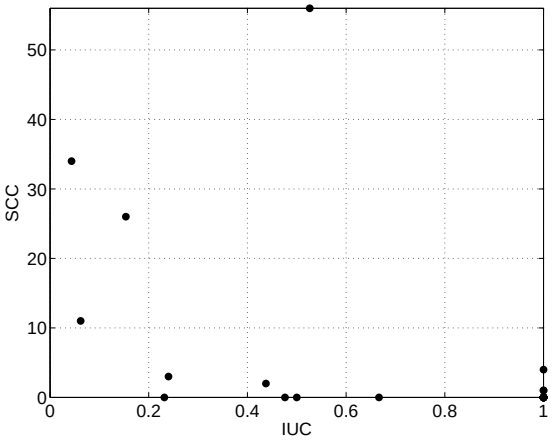


12: Lucene

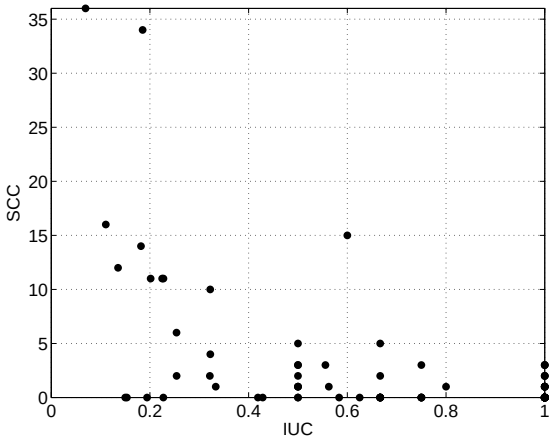
Figure D.1: IUC distribution scatter plots (2)



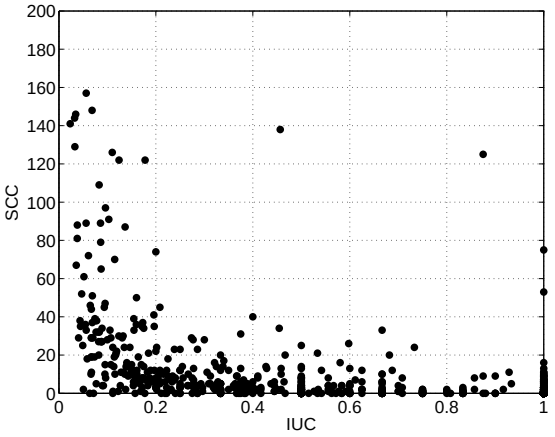
13: Rapidminer



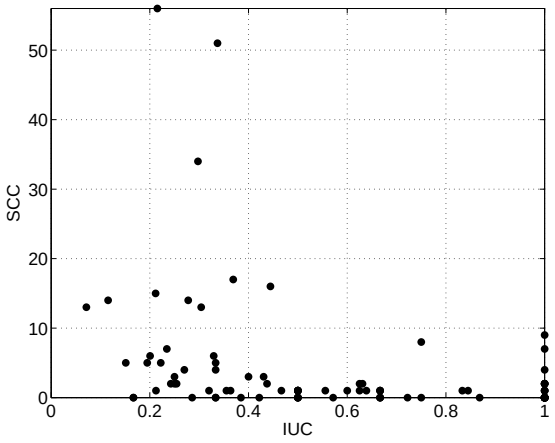
14: Sweet Home 3D



15: TripleA

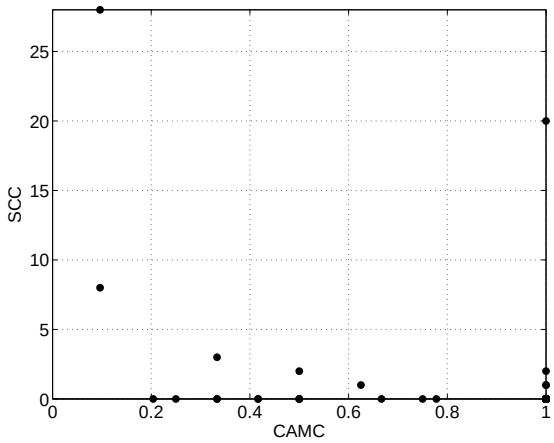


16: Vuze

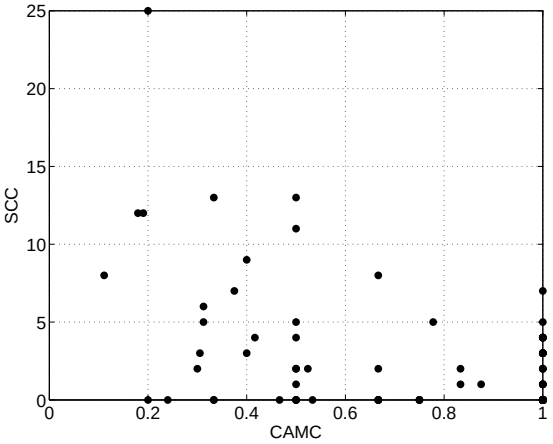


17: Xerces

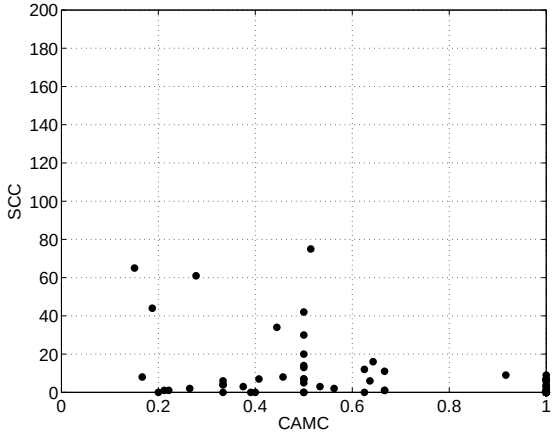
Figure D.1: IUC distribution scatter plots (3)



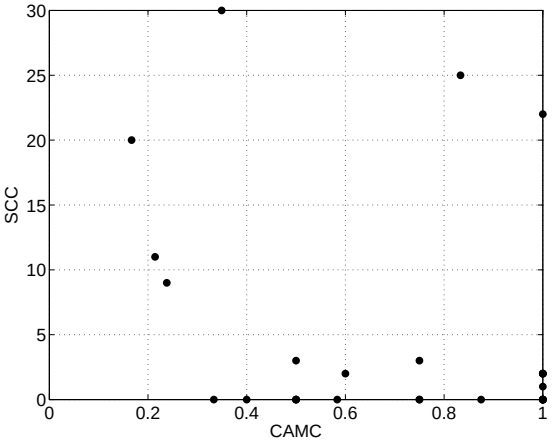
1: ArgoUML-app



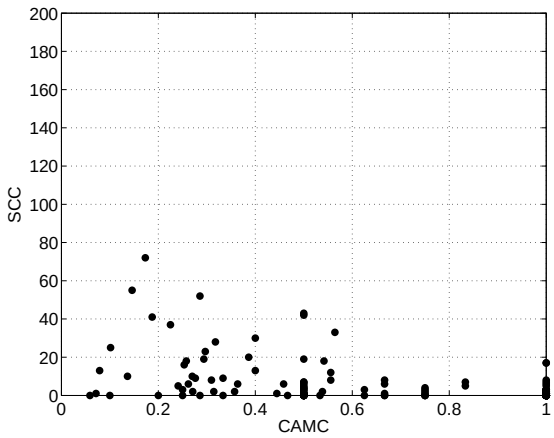
2: Eclipse.debug.ui



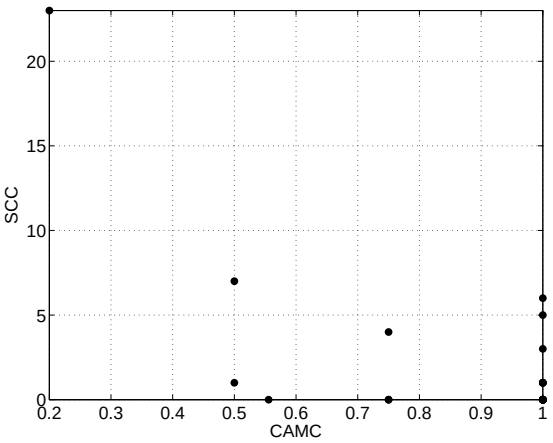
3: Eclipse.jdt.core



4: Eclipse.team.core

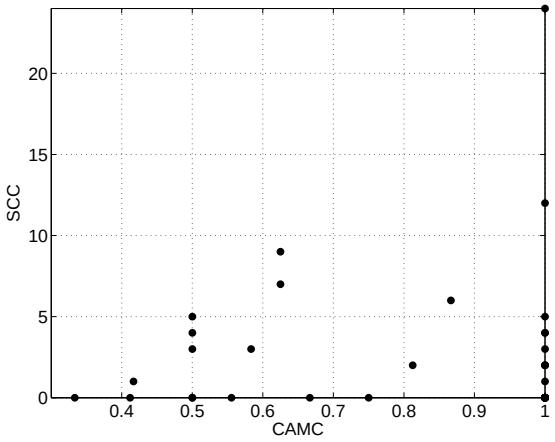


5: Hibernate3

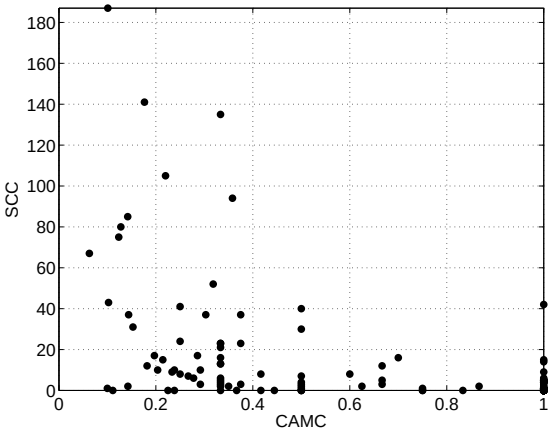


6: Jabref

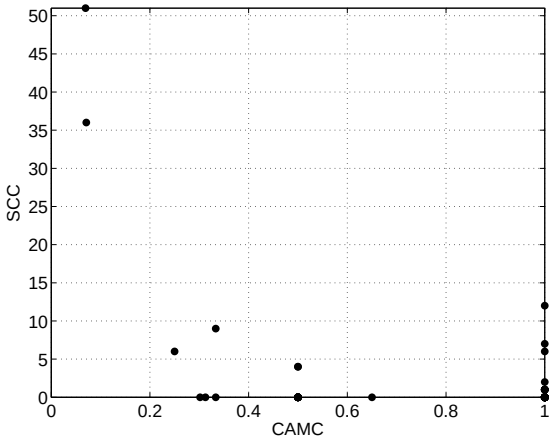
Figure D.2: CAMC distribution scatter plots



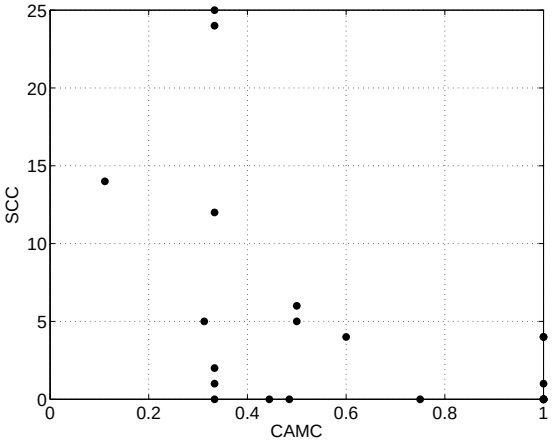
7: Jedit



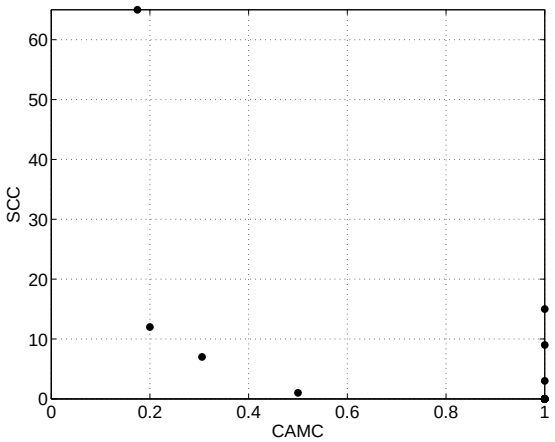
8: Jena2



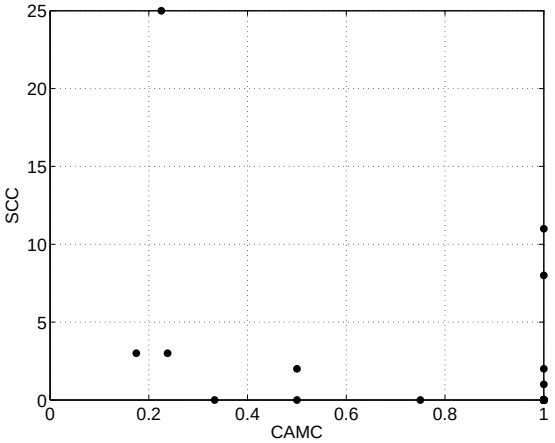
9: JFreeChart



10: Joone

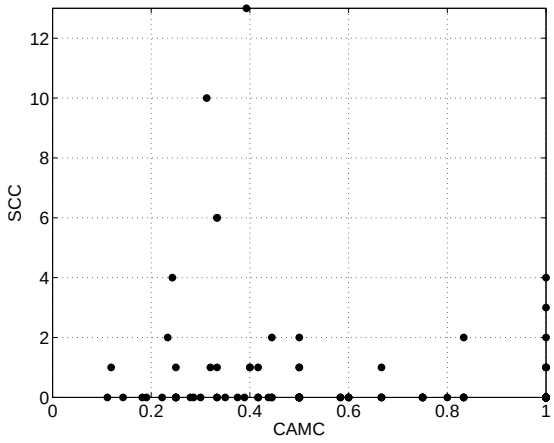


11: Log4j

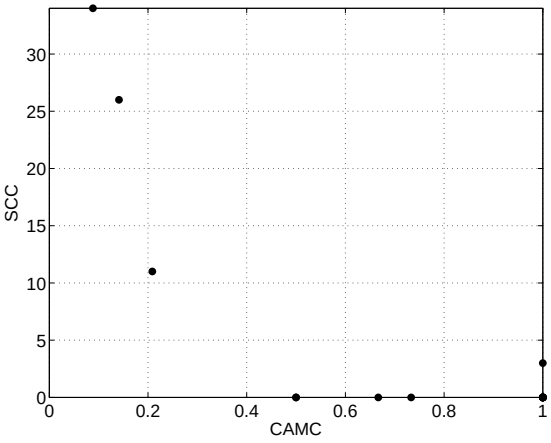


12: Lucene

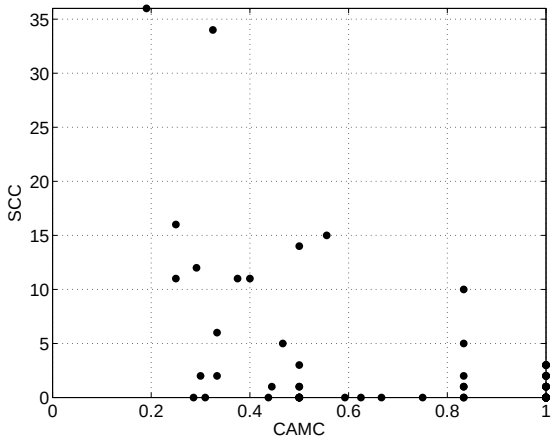
Figure D.2: CAMC distribution scatter plots (2)



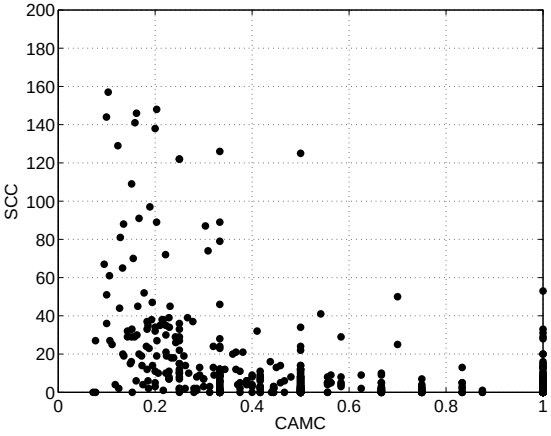
13: Rapidminer



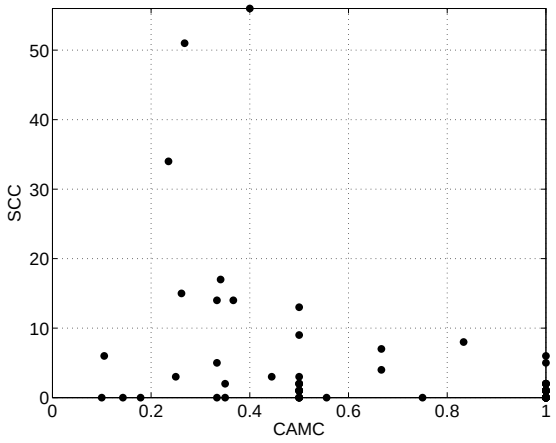
14: Sweet Home 3D



15: TripleA

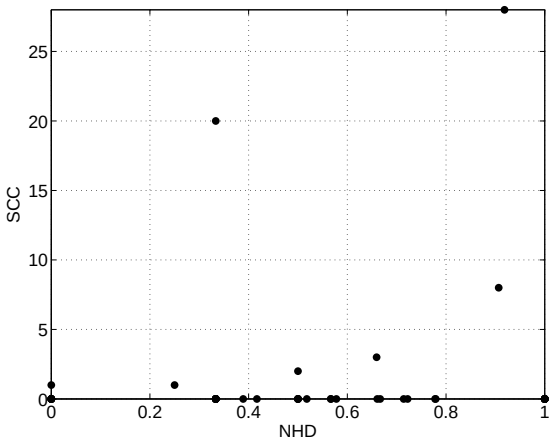


16: Vuze

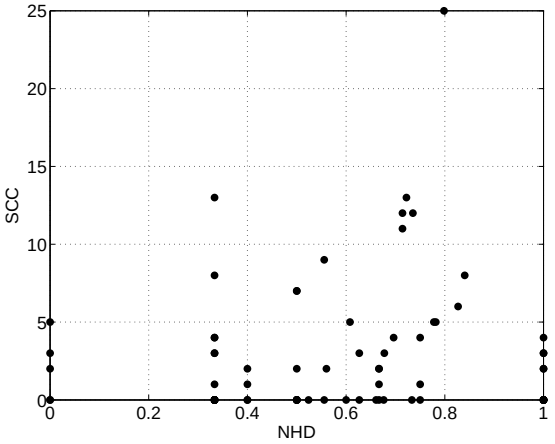


17: Xerces

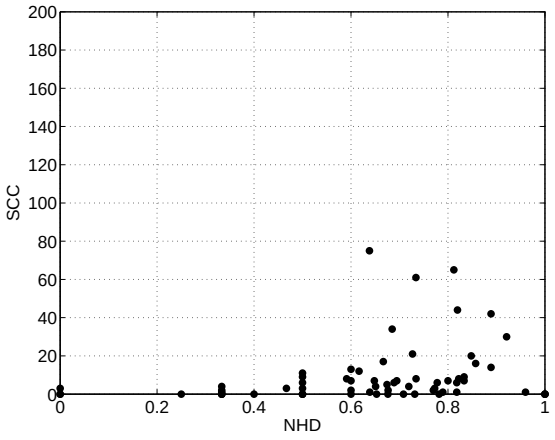
Figure D.2: CAMC distribution scatter plots (3)



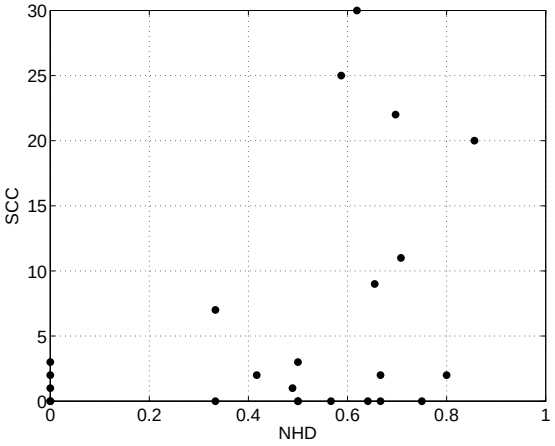
1: ArgoUML-app



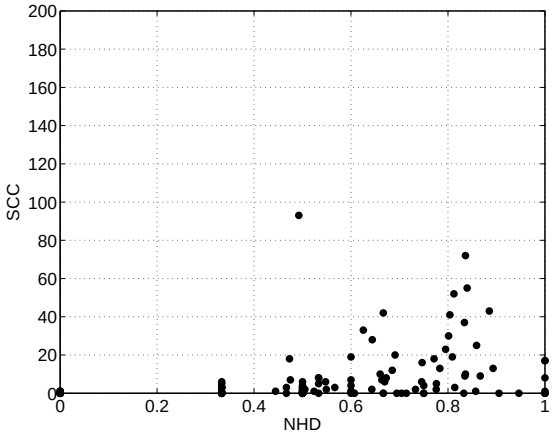
2: Eclipse.debug.ui



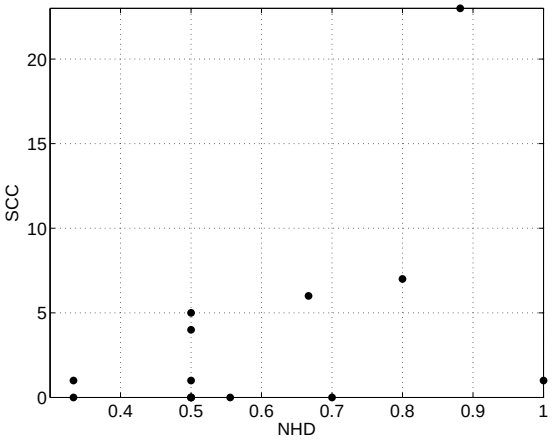
3: Eclipse.jdt.core



4: Eclipse.team.core

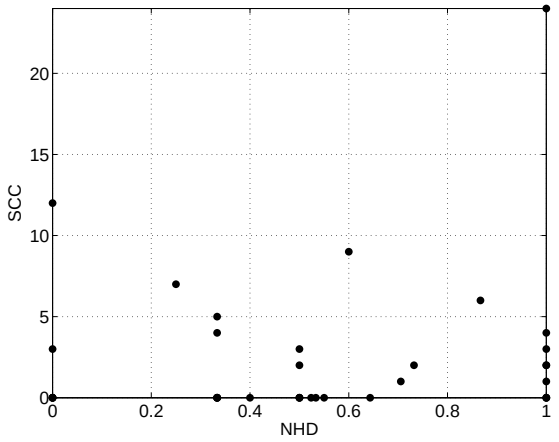


5: Hibernate3

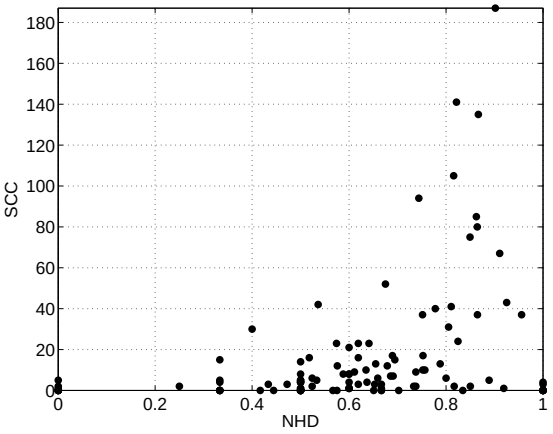


6: Jabref

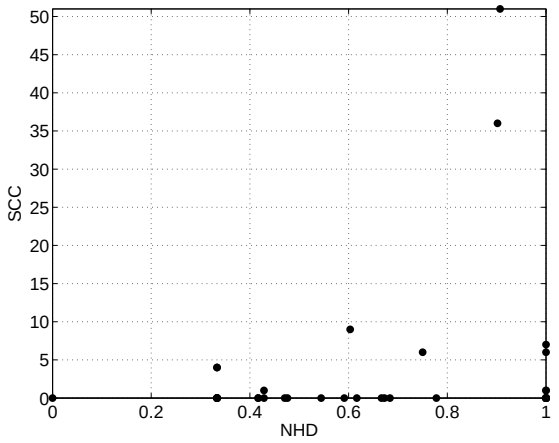
Figure D.3: NHD distribution scatter plots



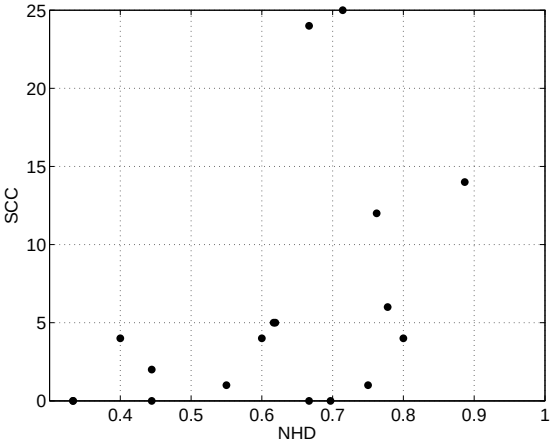
7: JEdit



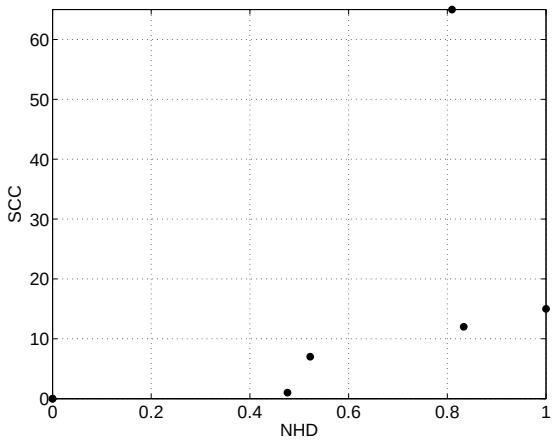
8: Jena2



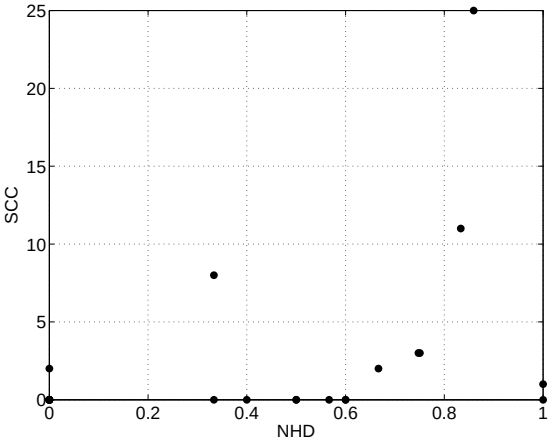
9: JFreeChart



10: Joone

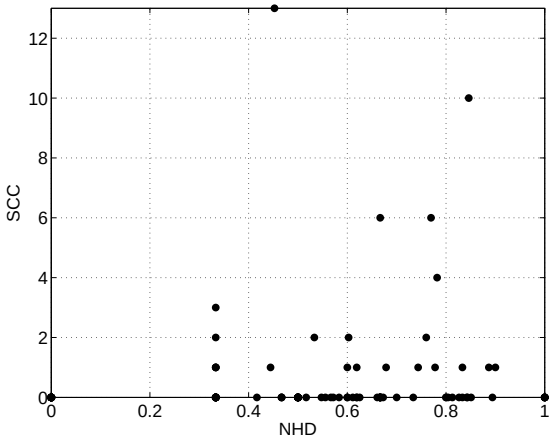


11: Log4j

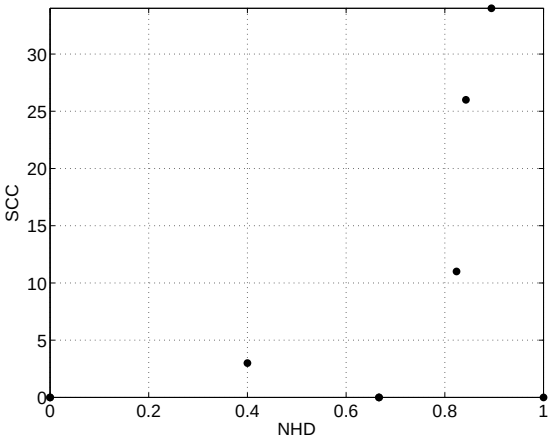


12: Lucene

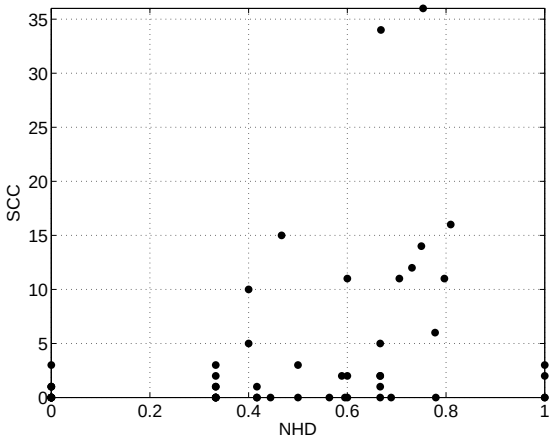
Figure D.3: NHD distribution scatter plots (2)



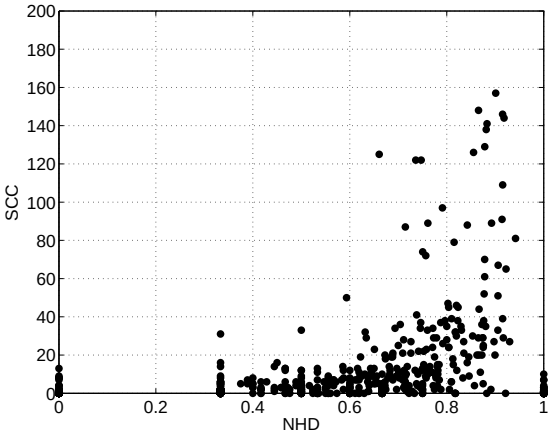
13: Rapidminer



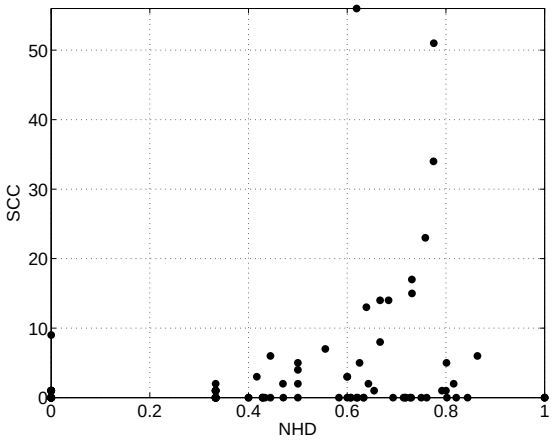
14: Sweet Home 3D



15: TripleA

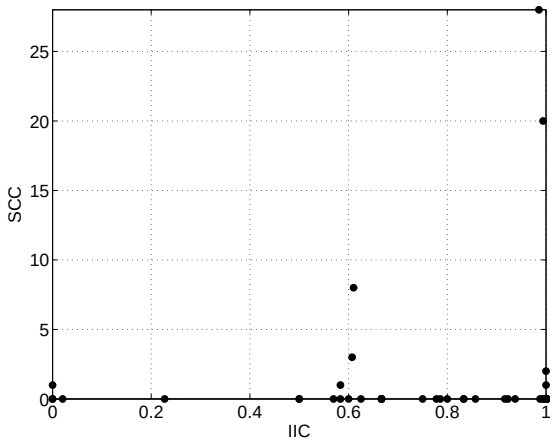


16: Vuze

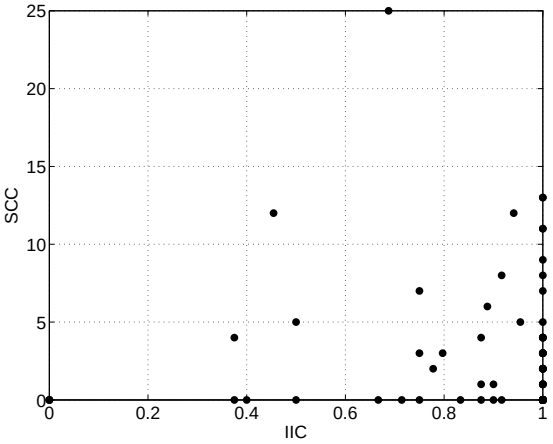


17: Xerces

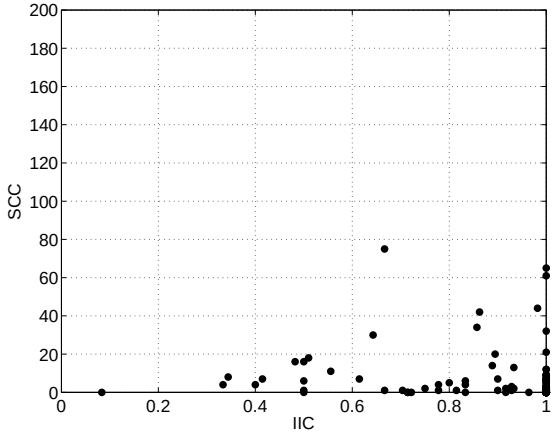
Figure D.3: NHD distribution scatter plots (3)



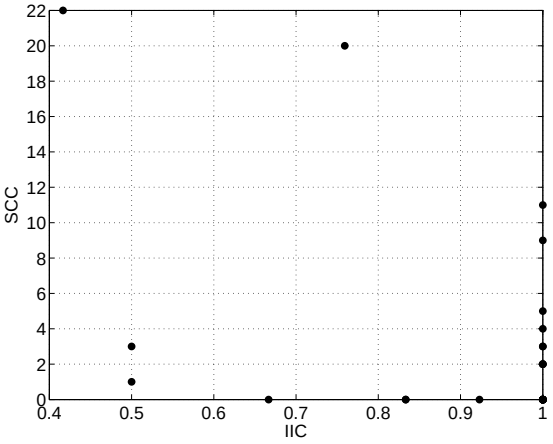
1: ArgoUML-app



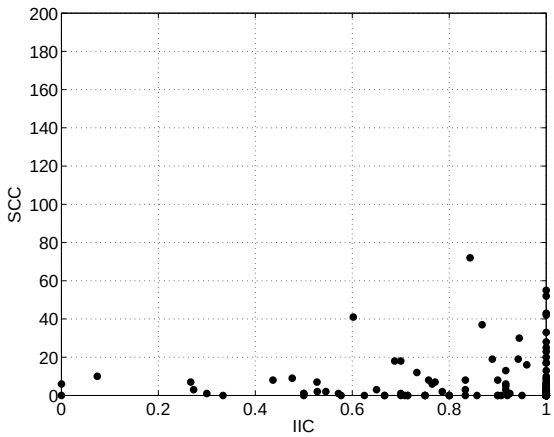
2: Eclipse.debug.ui



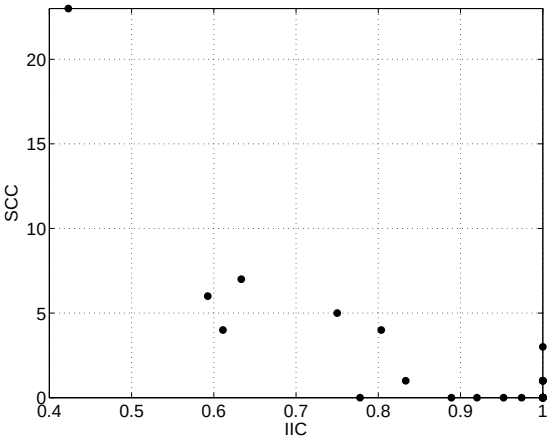
3: Eclipse.jdt.core



4: Eclipse.team.core

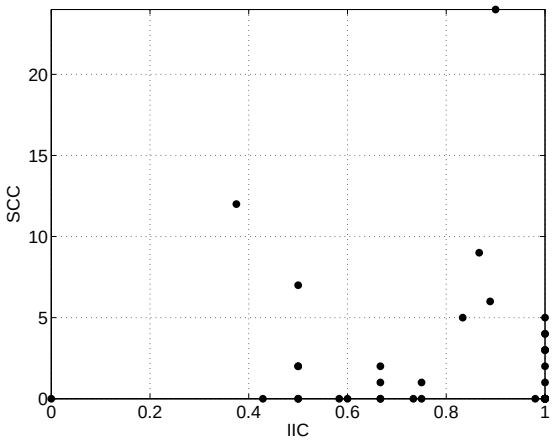


5: Hibernate3

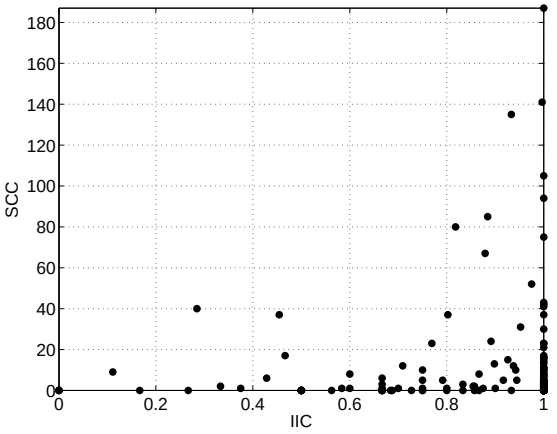


6: Jabref

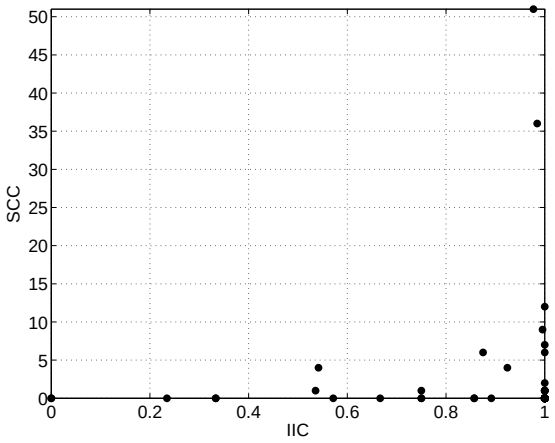
Figure D.4: IIC distribution scatter plots



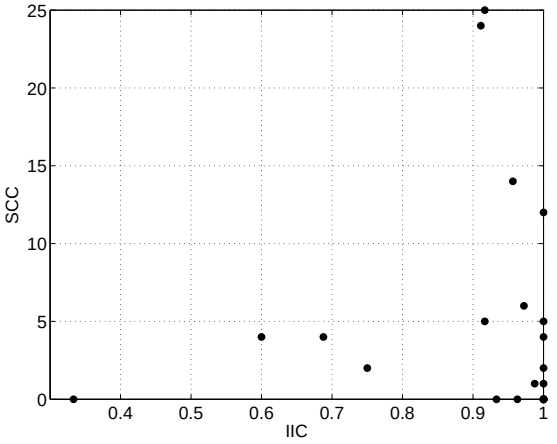
7: JEdit



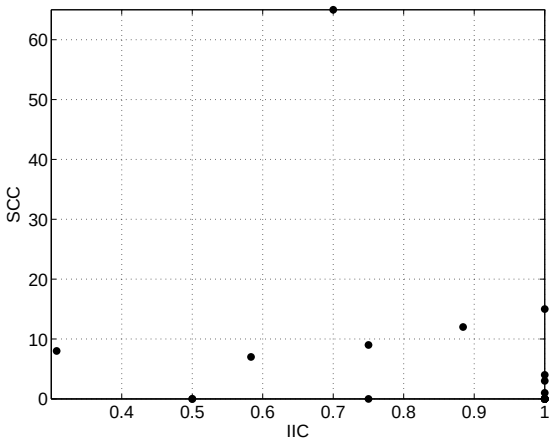
8: Jena2



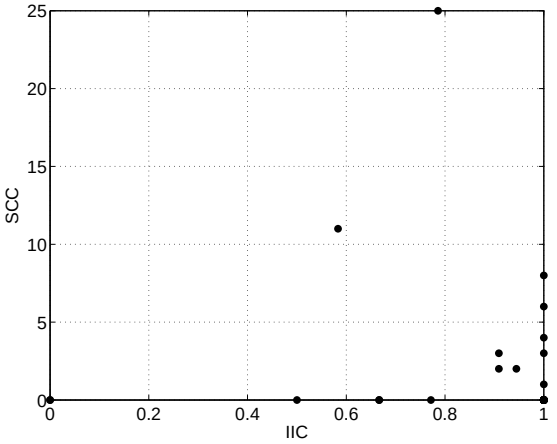
9: JFreeChart



10: Joone

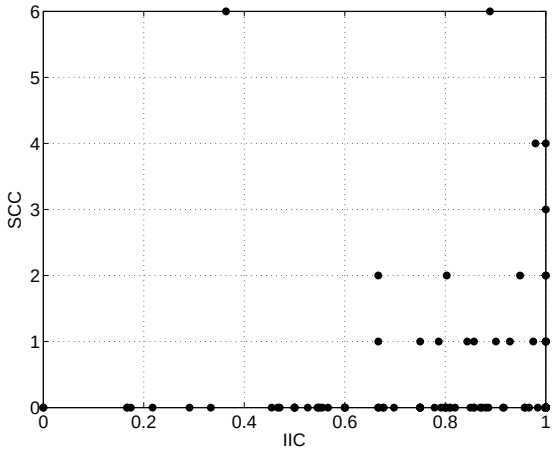


11: Log4j

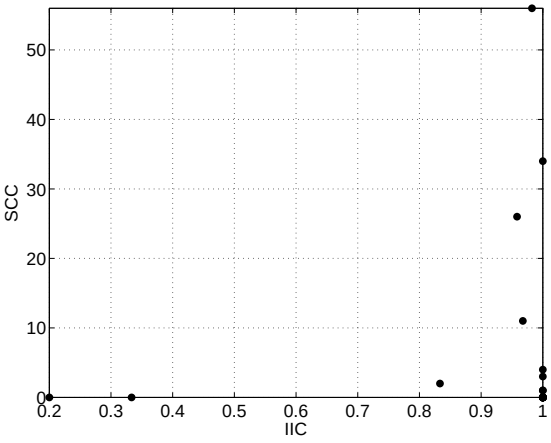


12: Lucene

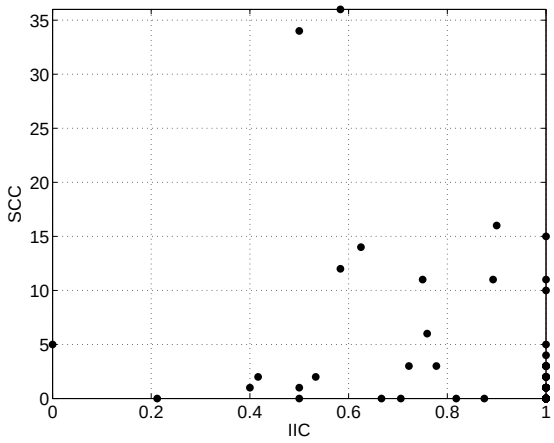
Figure D.4: IIC distribution scatter plots (2)



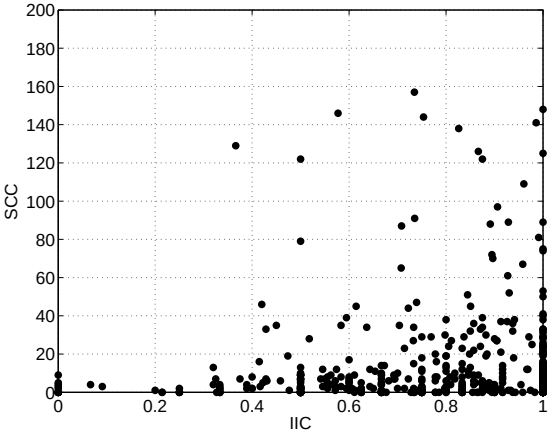
13: Rapidminer



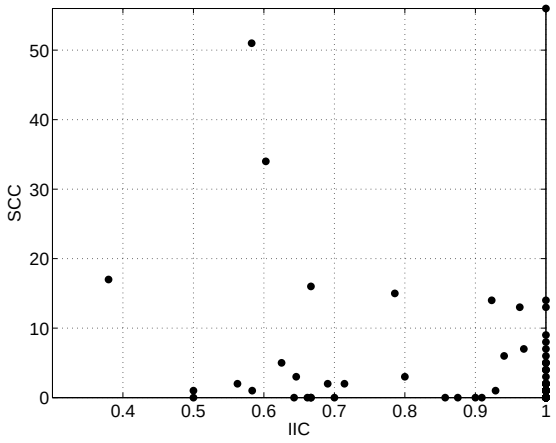
14: Sweet Home 3D



15: TripleA

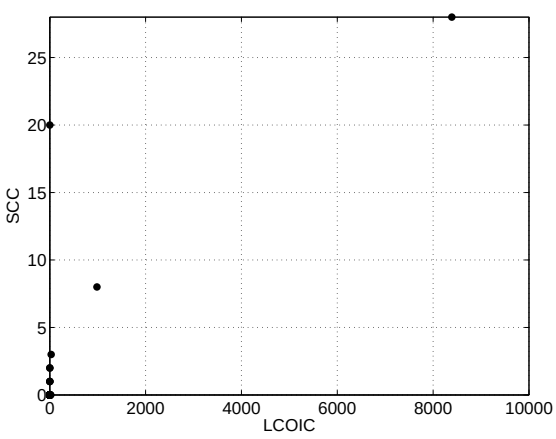


16: Vuze2

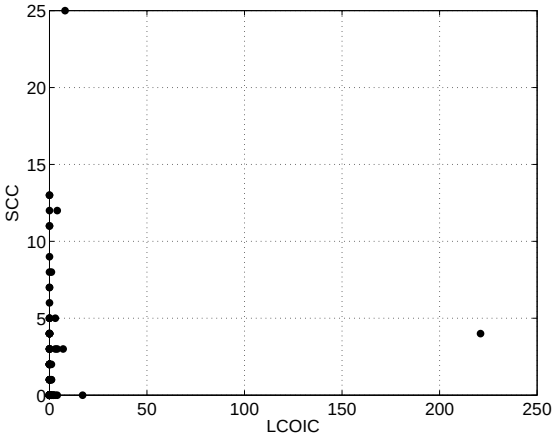


17: Xerces

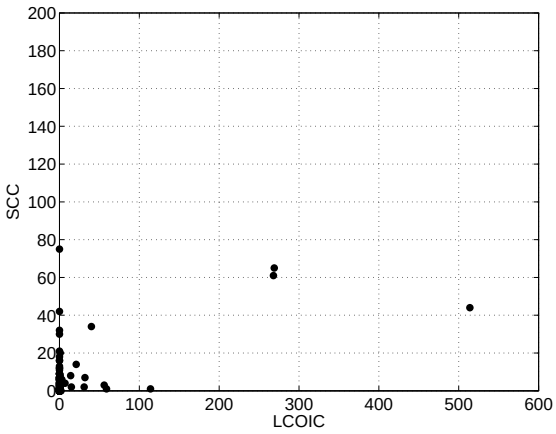
Figure D.4: IIC distribution scatter plots (3)



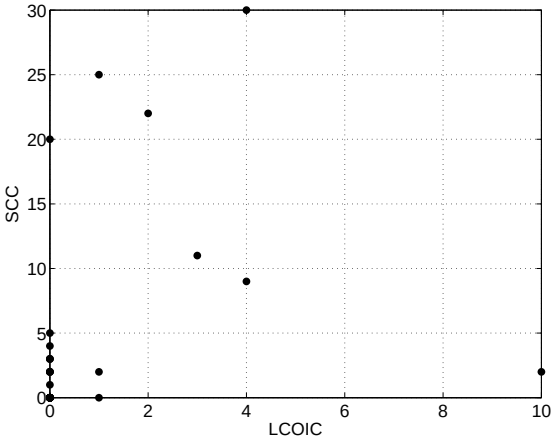
1: ArgoUML-app



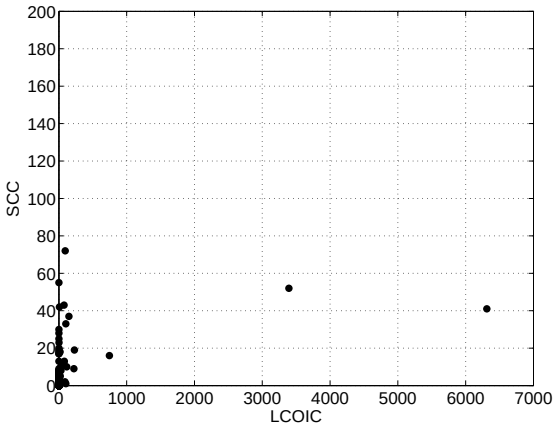
2: Eclipse.debug.ui



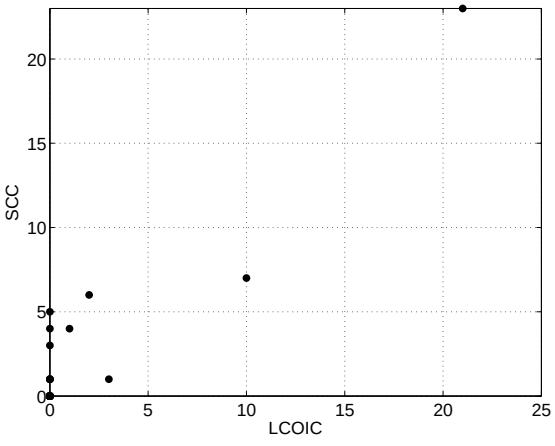
3: Eclipse.jdt.core



4: Eclipse.team.core

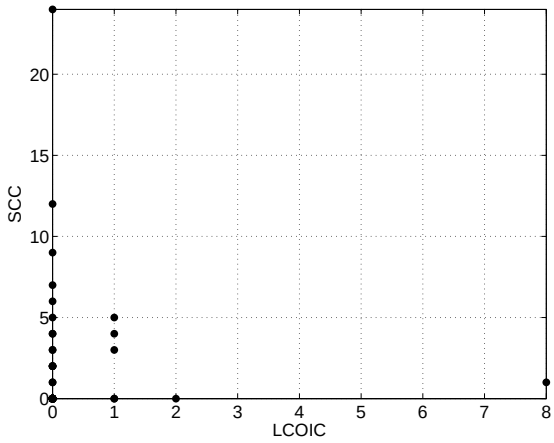


5: Hibernate3

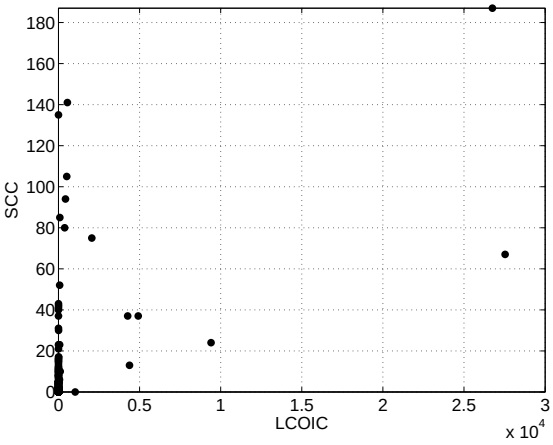


6: Jabref

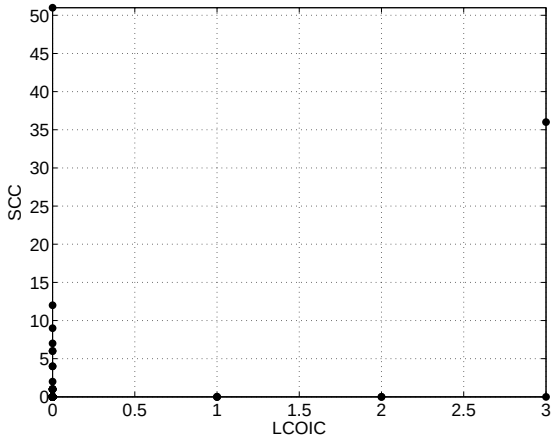
Figure D.5: LCOIC distribution scatter plots



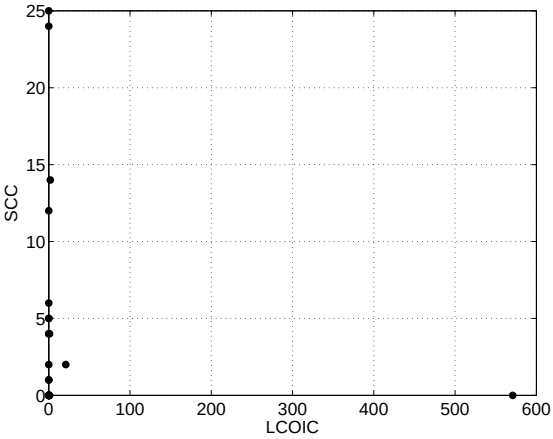
7: JEdit



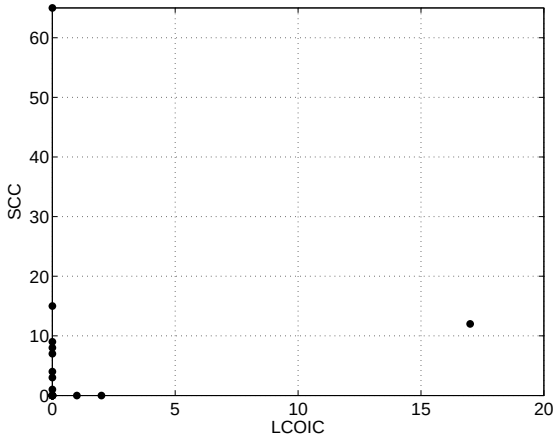
8: Jena2



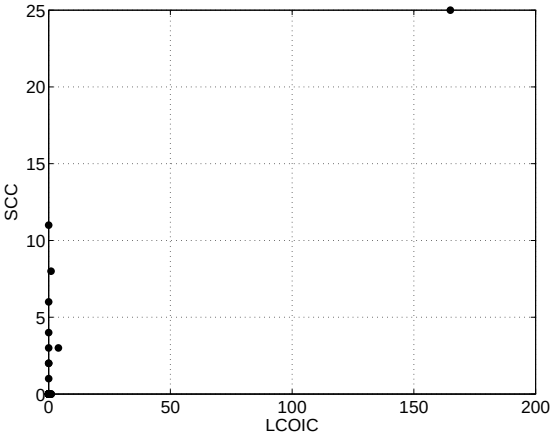
9: JFreeChart



10: Joone

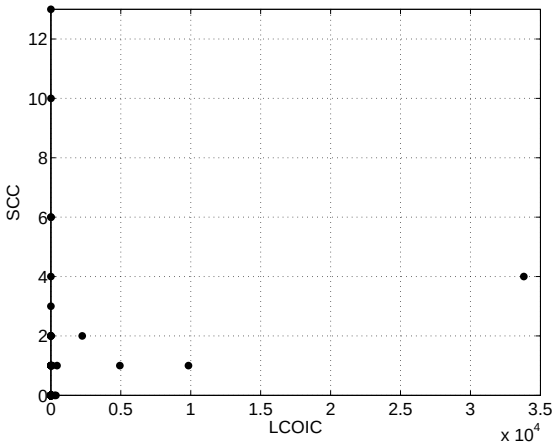


11: Log4j

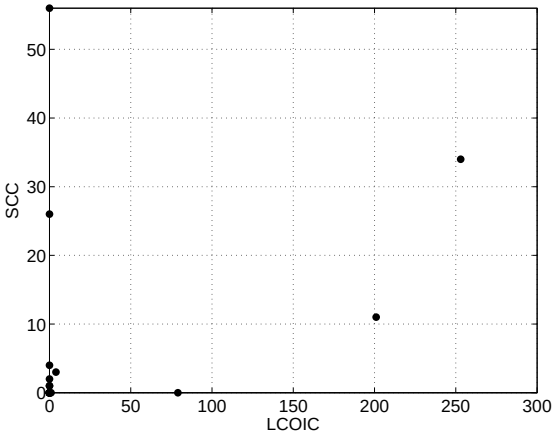


12: Lucene

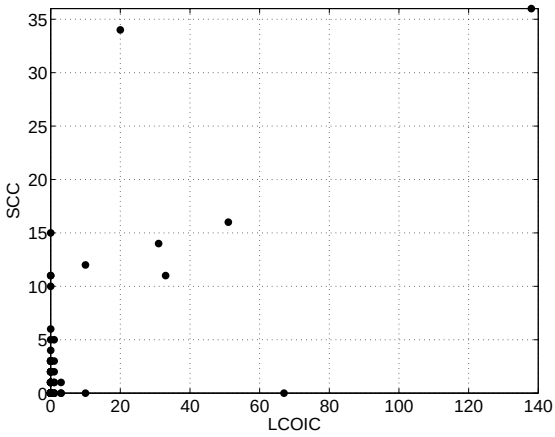
Figure D.5: LCOIC distribution scatter plots (2)



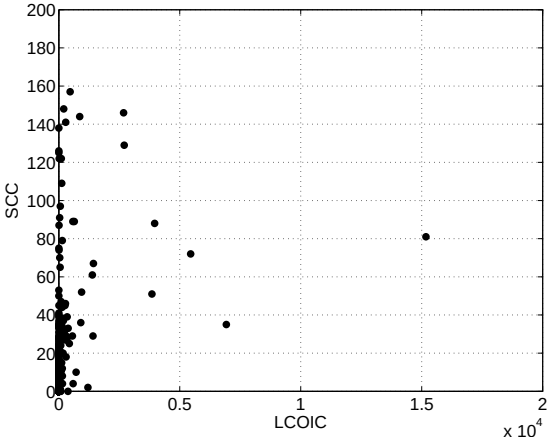
13: RapidMiner



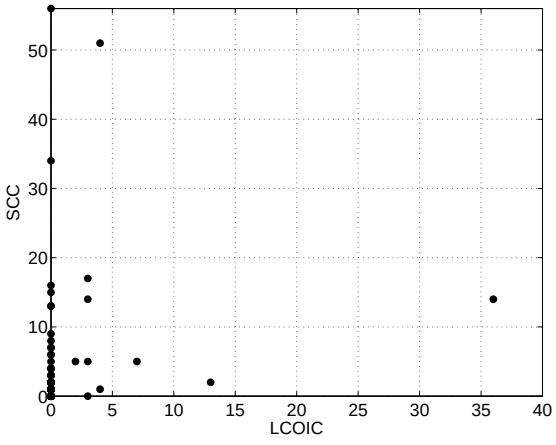
14: Sweet Home 3D



15: TripleA



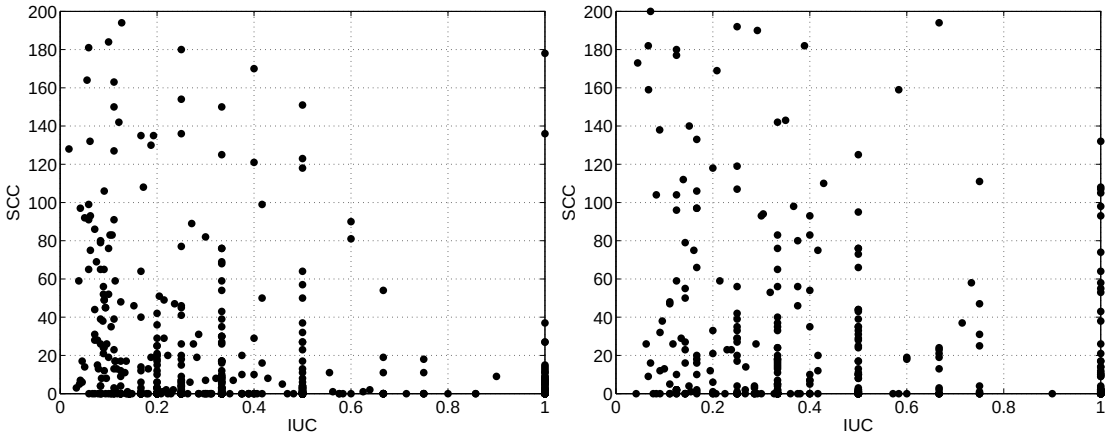
16: Vuze



17: Xerces

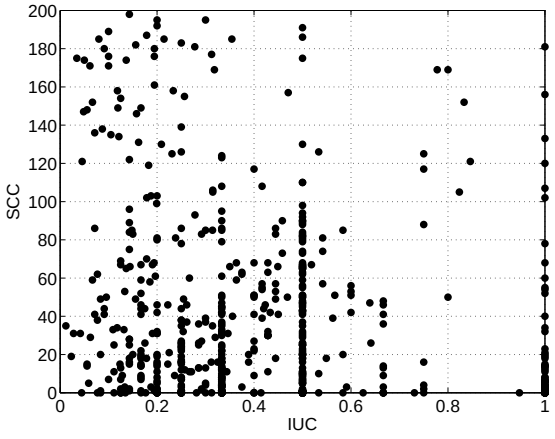
Figure D.5: LCOIC distribution scatter plots (3)

D.2.2 Scatter plots for concrete classes

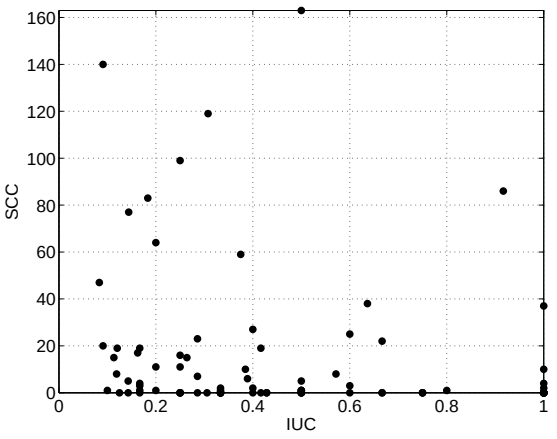


1: ArgoUML

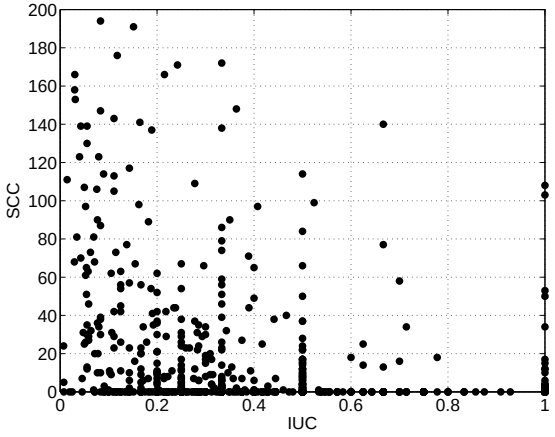
2: Eclipse.debug.ui



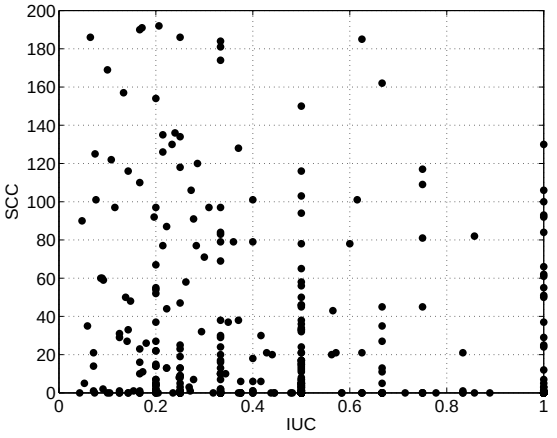
3: Eclipse.jdt.core



4: Eclipse.team.core

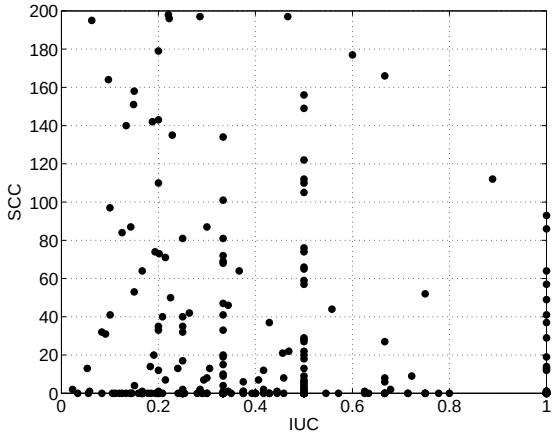


5: Hibernate3

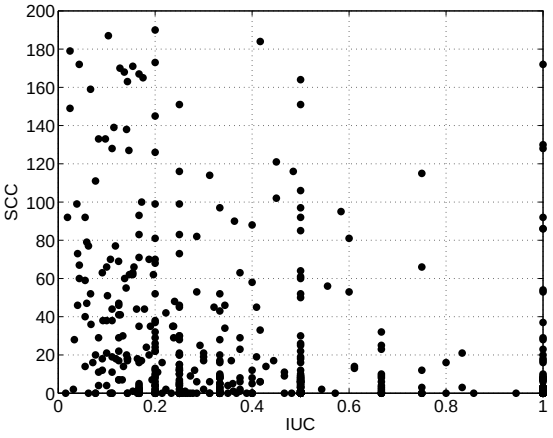


6: Jabref

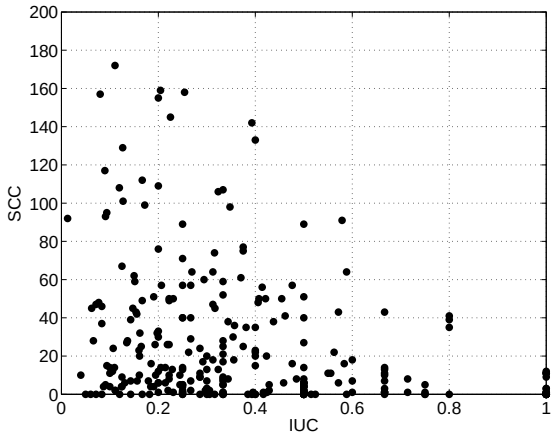
Figure D.6: IUC distribution scatter plots



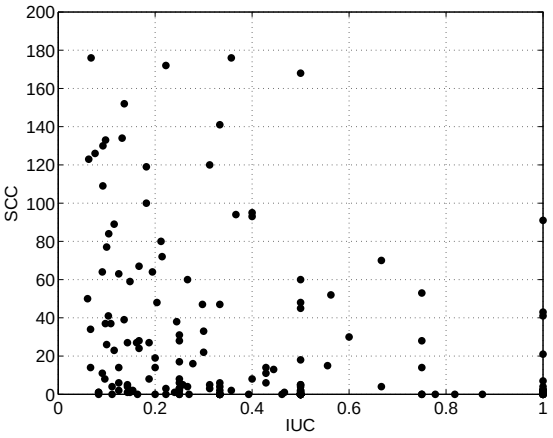
7: Jedit



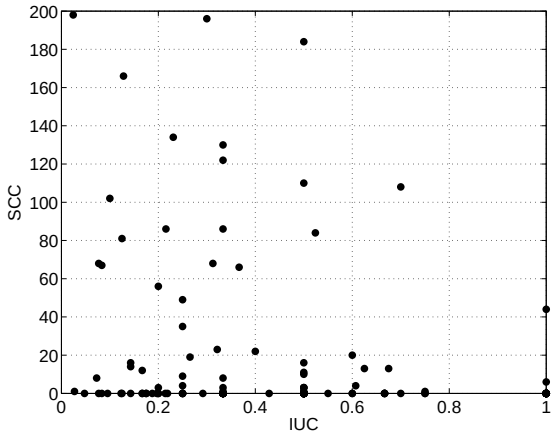
8: Jena2



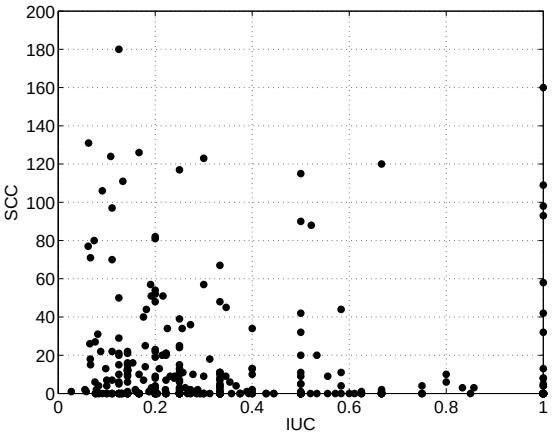
9: JFreeChart



10: Joone

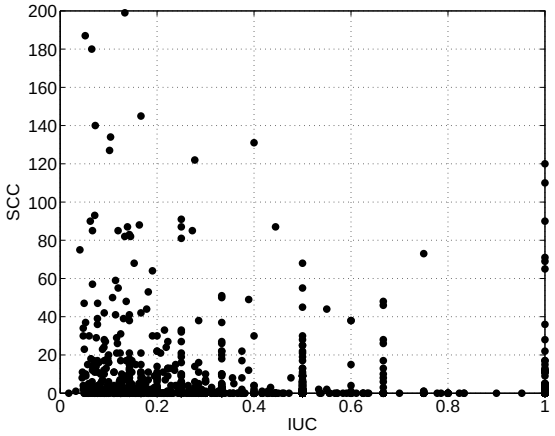


11: Log4j

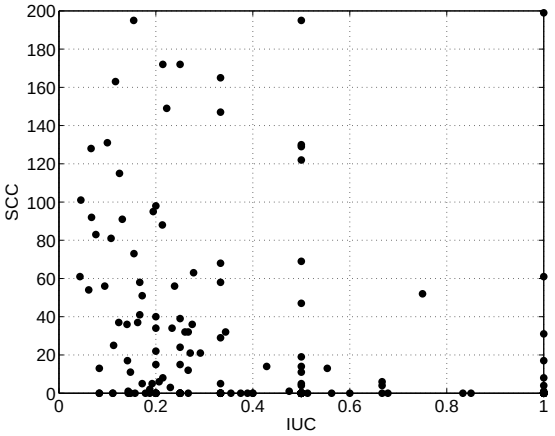


12: Lucene

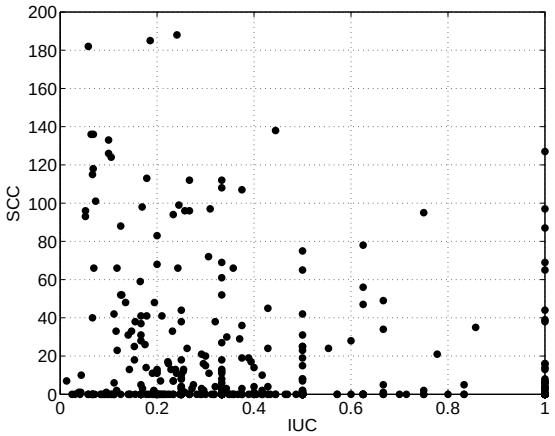
Figure D.6: IUC distribution scatter plots (2)



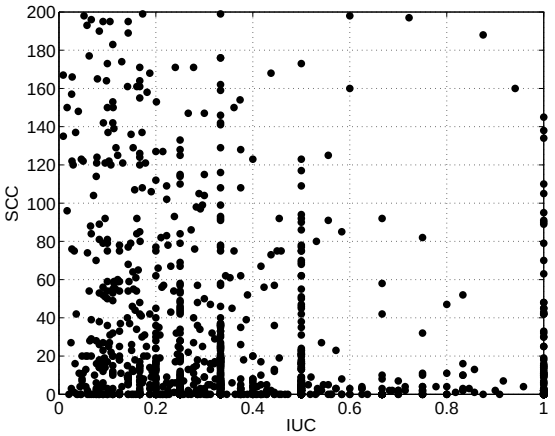
13: Rapidminer



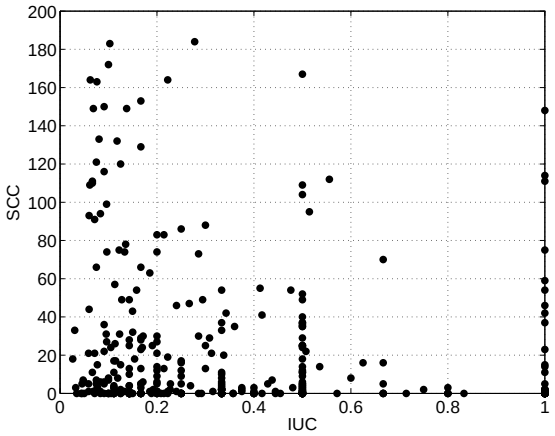
14: Sweet Home 3D



15: TripleA

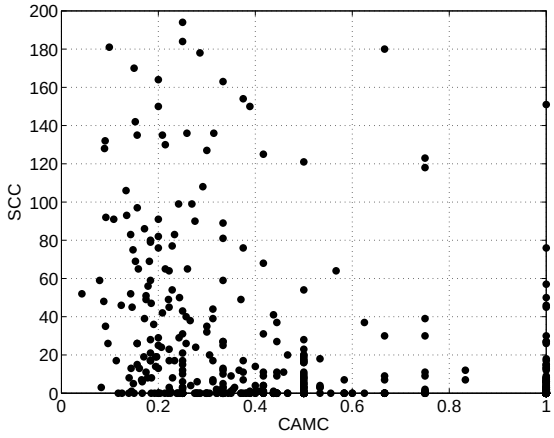


16: Vuze

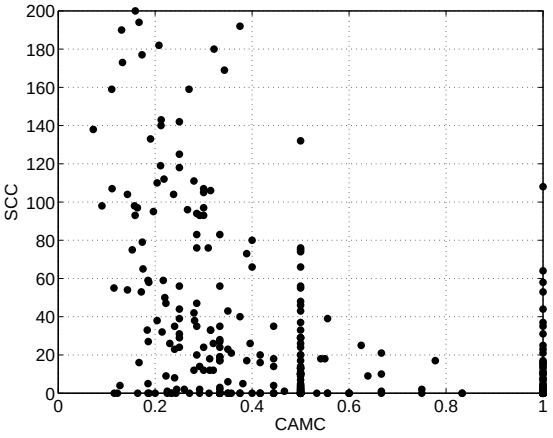


17: Xerces

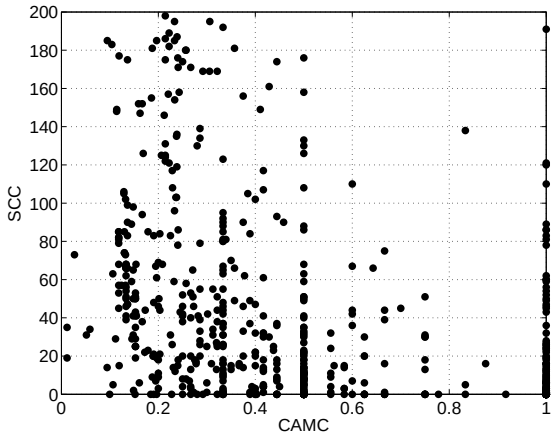
Figure D.6: IUC distribution scatter plots (3)



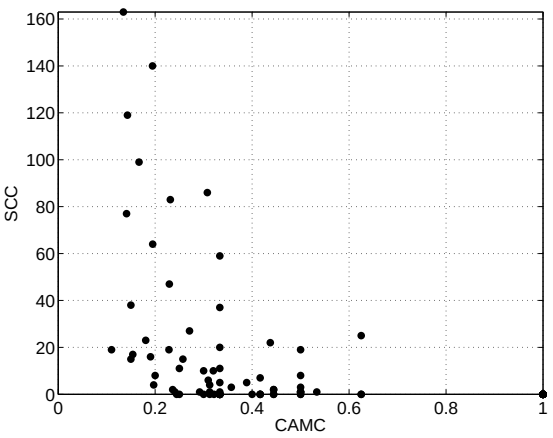
1: ArgoUML



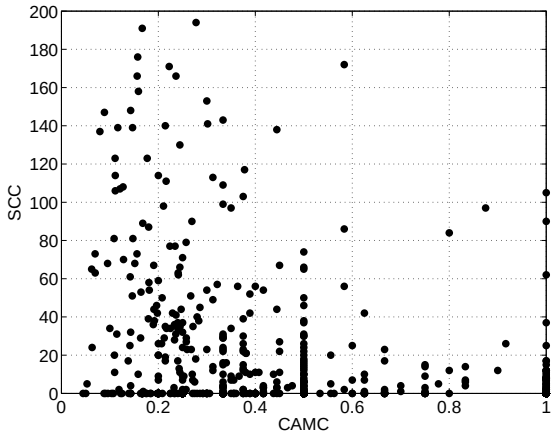
2: Eclipse.debug.ui



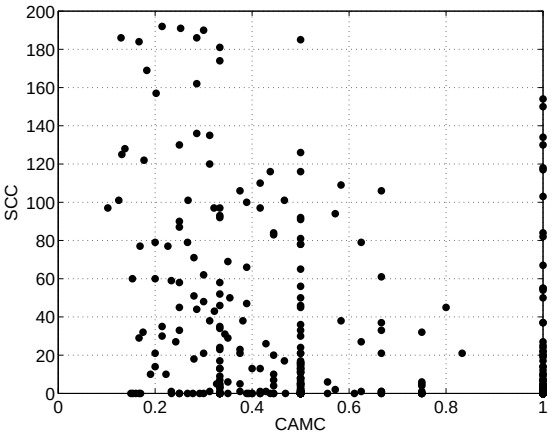
3: Eclipse.jdt.core



4: Eclipse.team.core

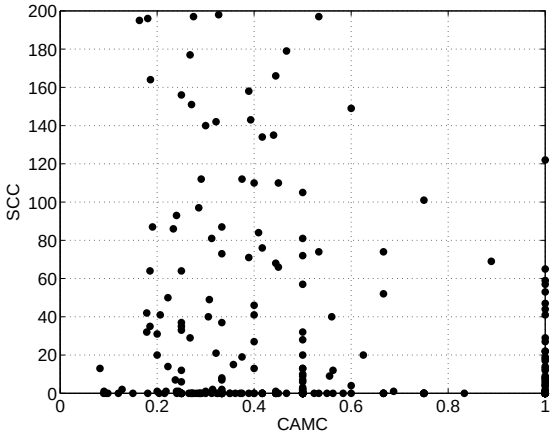


5: Hibernate3

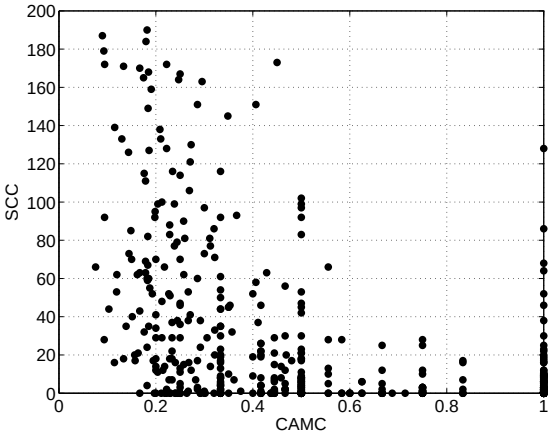


6: Jabref

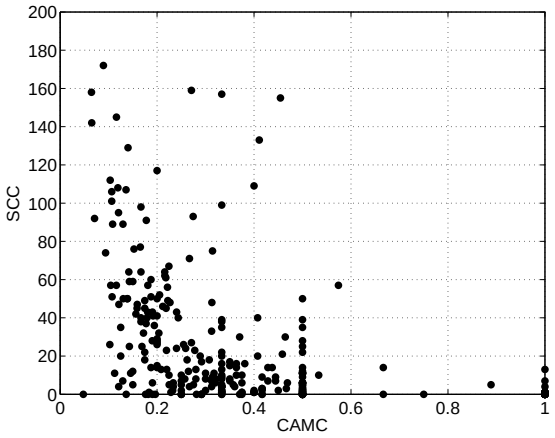
Figure D.7: CAMC distribution scatter plots



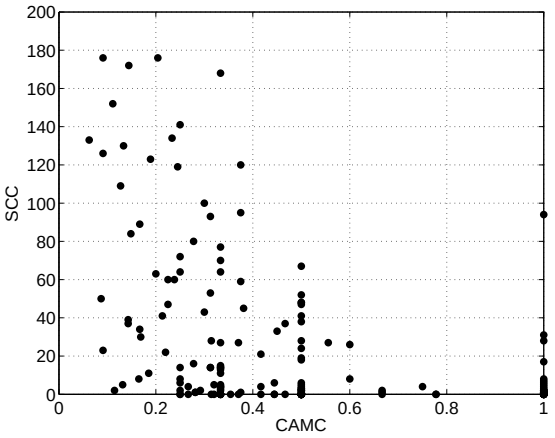
7: Jedit



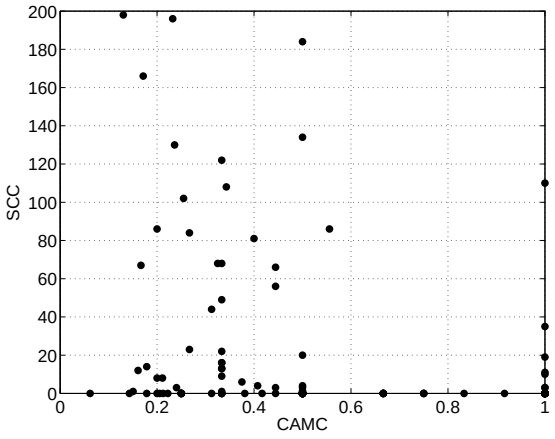
8: Jena2



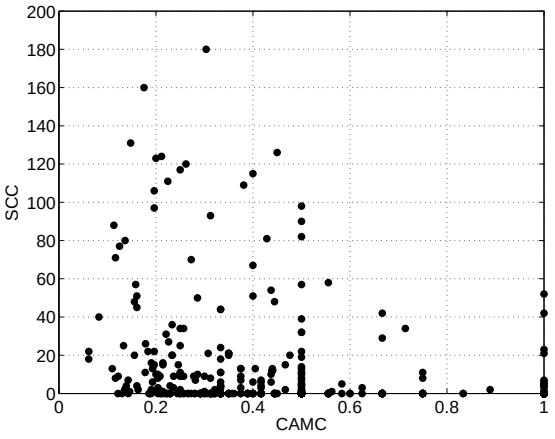
9: JFreeChart



10: Joone

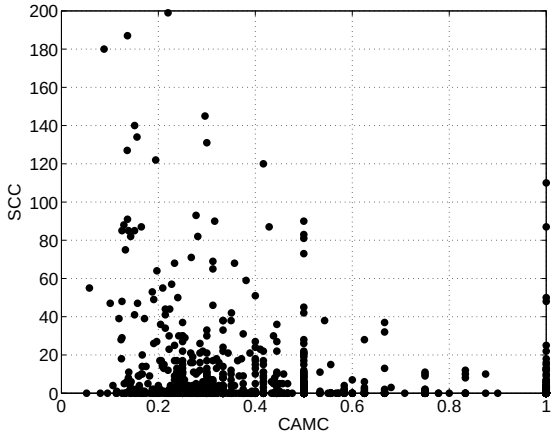


11: Log4j

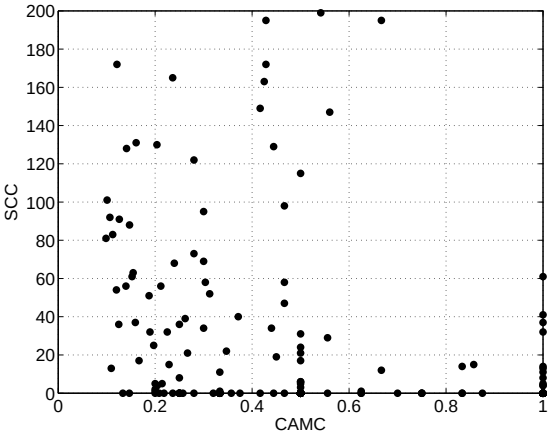


12: Lucene

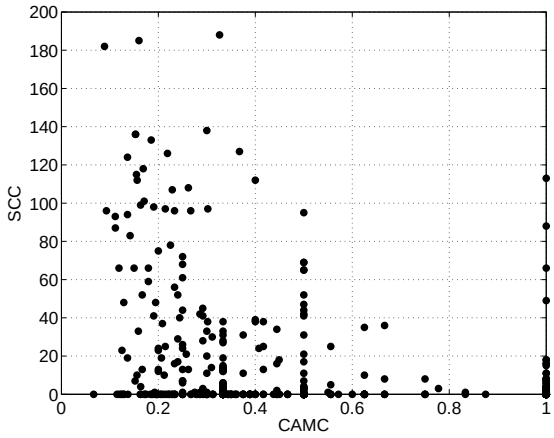
Figure D.7: CAMC distribution scatter plots (2)



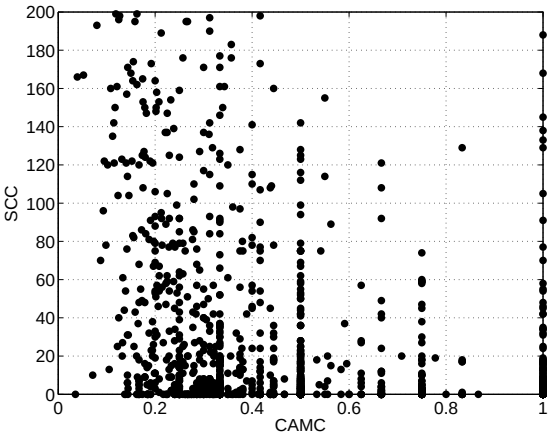
13: Rapidminer



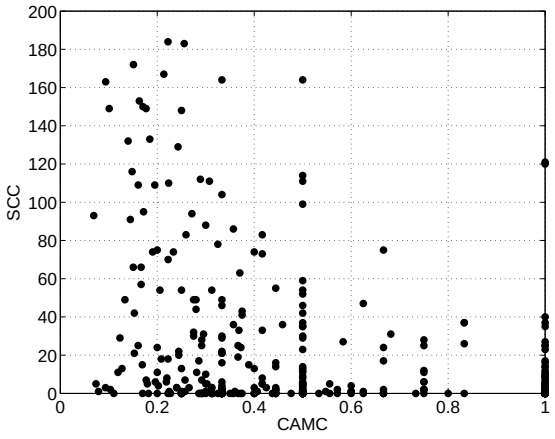
14: Sweet Home 3D



15: TripleA

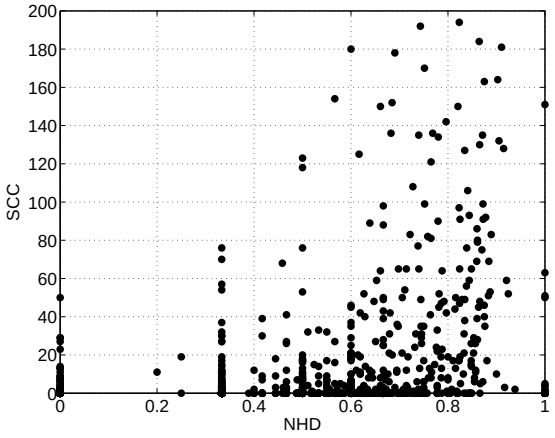


16: Vuze2

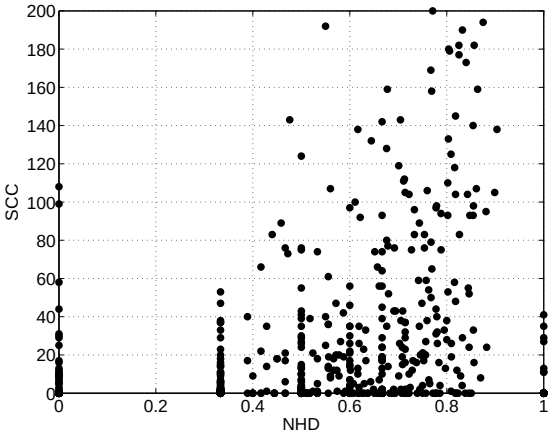


17: Xerces

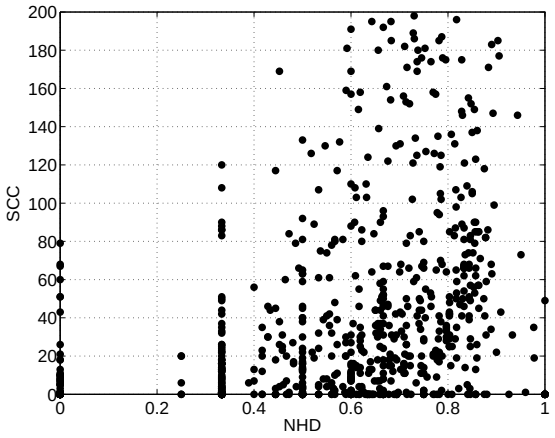
Figure D.7: CAMC distribution scatter plots (3)



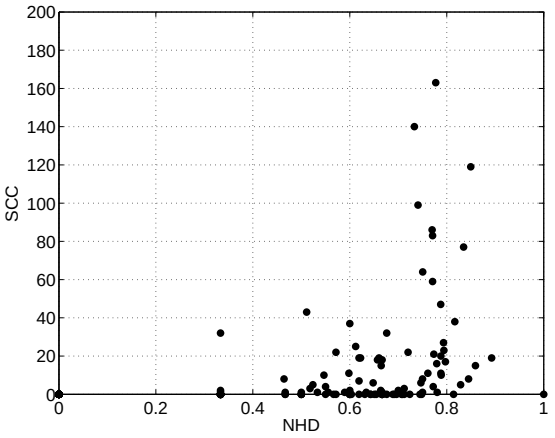
1: ArgoUML-app



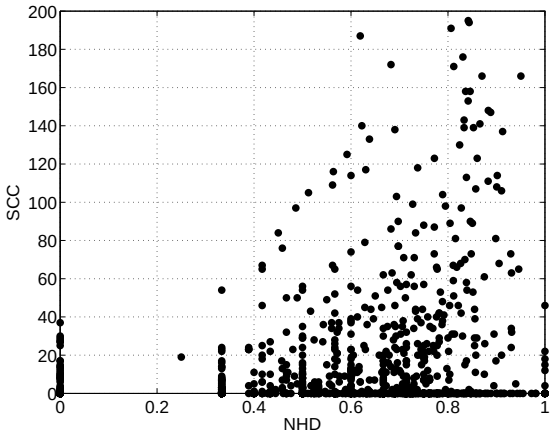
2: Eclipse.debug.ui



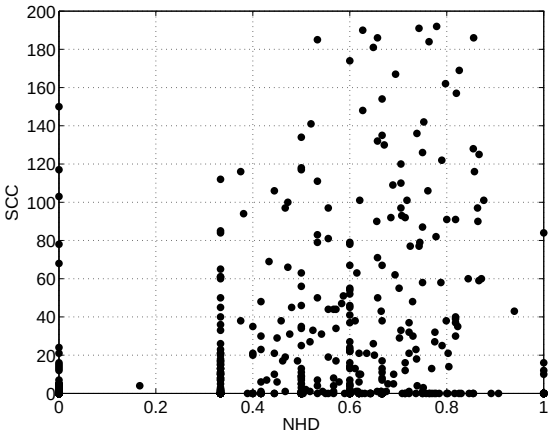
3: Eclipse.jdt.core



4: Eclipse.team.core

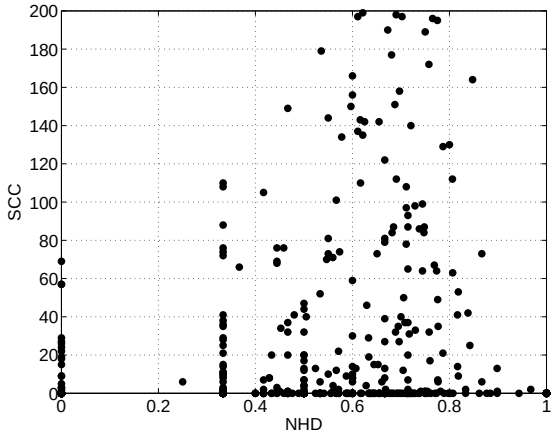


5: Hibernate3

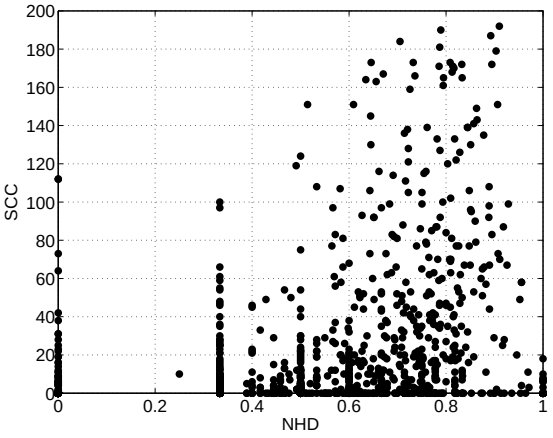


6: JabRef

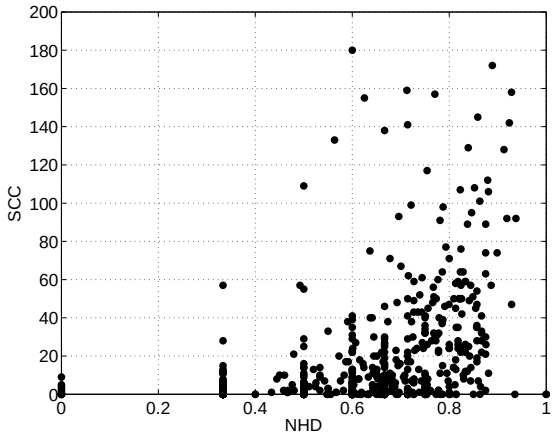
Figure D.8: NHD distribution scatter plots



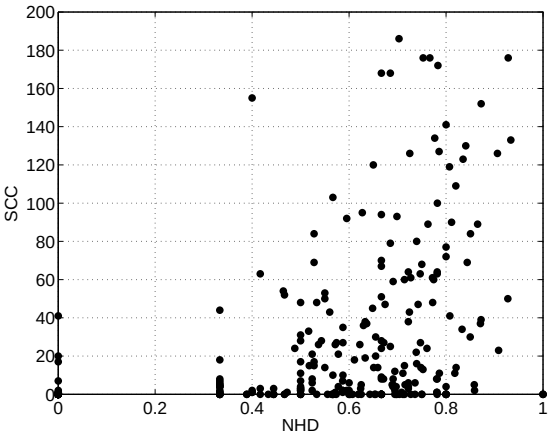
7: JEdit



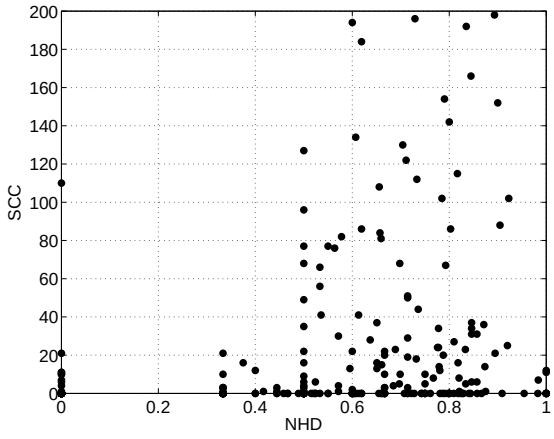
8: Jena2



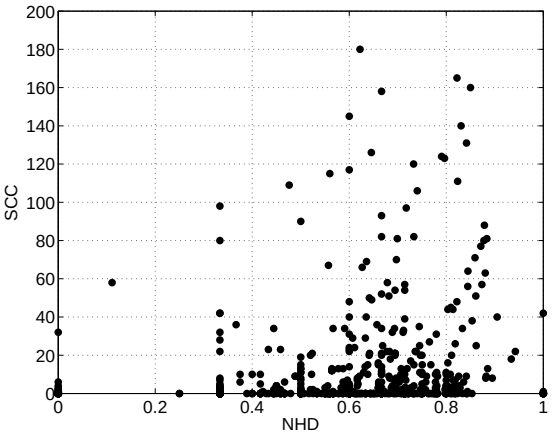
9: JFreeChart



10: Joone

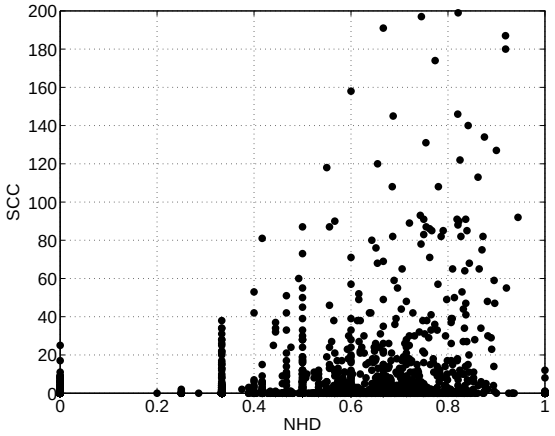


11: Log4j

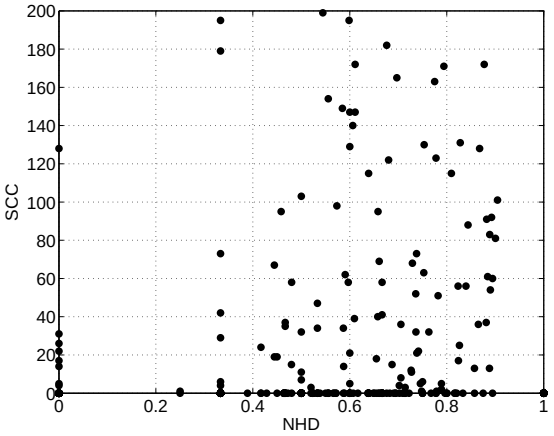


12: Lucene

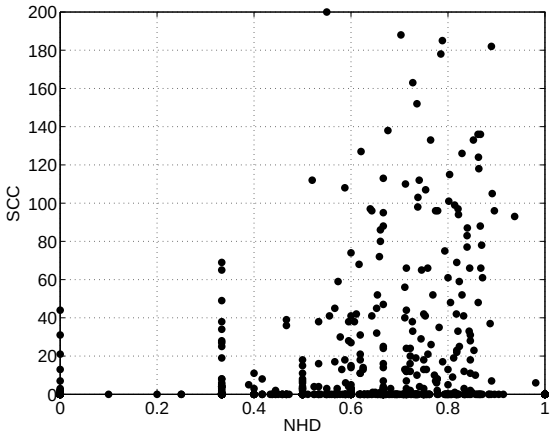
Figure D.8: NHD distribution scatter plots (2)



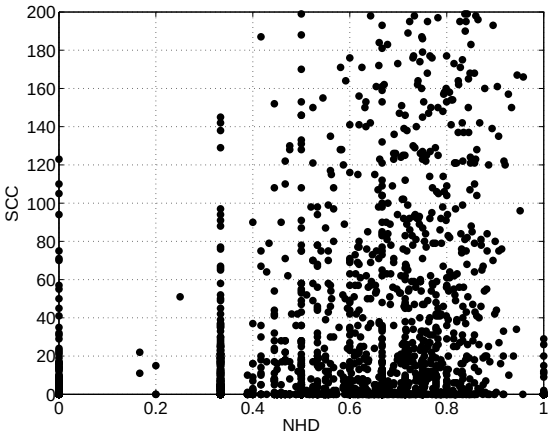
13: Rapidminer



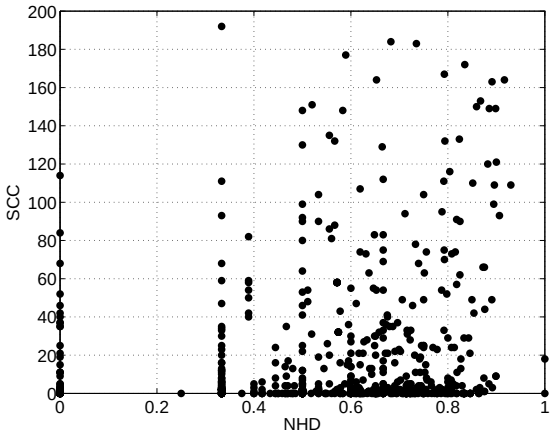
14: Sweet Home 3D



15: TripleA

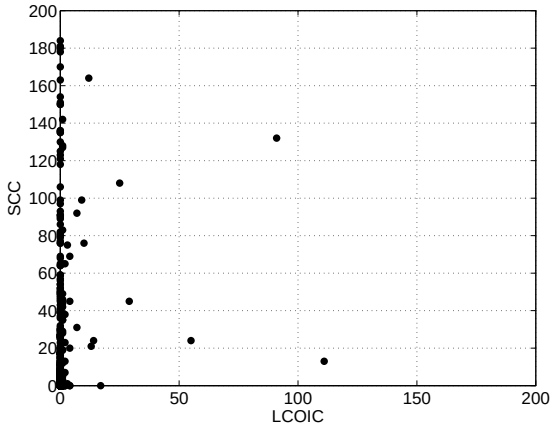


16: Vuze

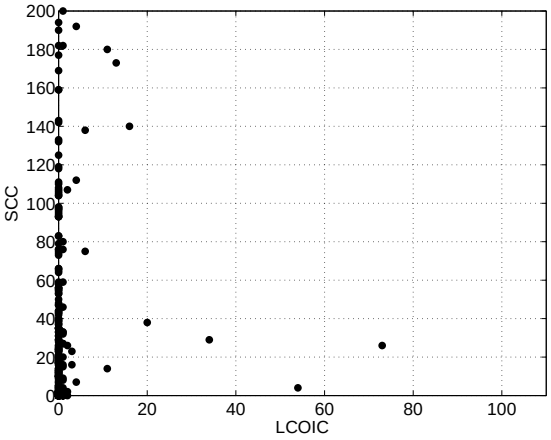


17: Xerces

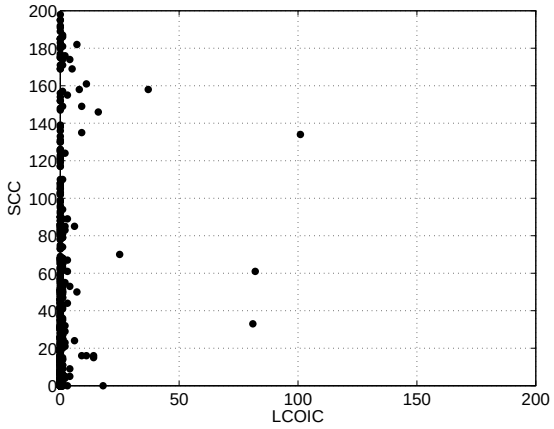
Figure D.8: NHD distribution scatter plots (3)



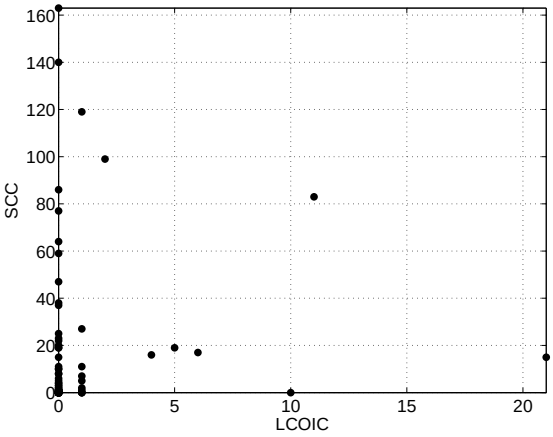
1: ArgoUML



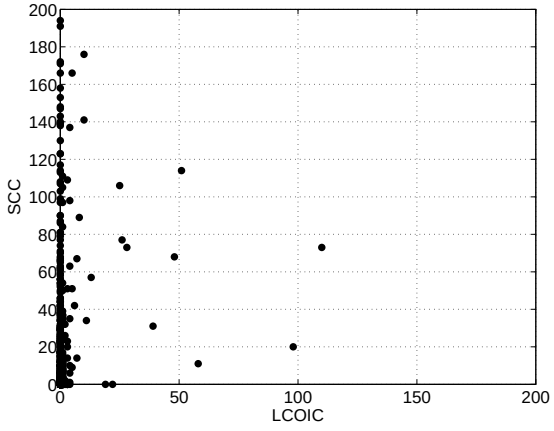
2: Eclipse.debug.ui



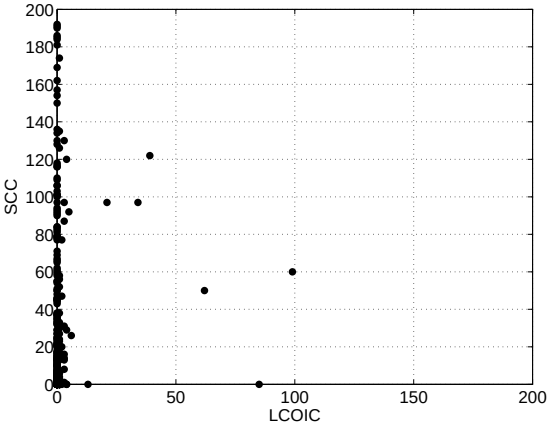
3: Eclipse.jdt.core



4: Eclipse.team.core

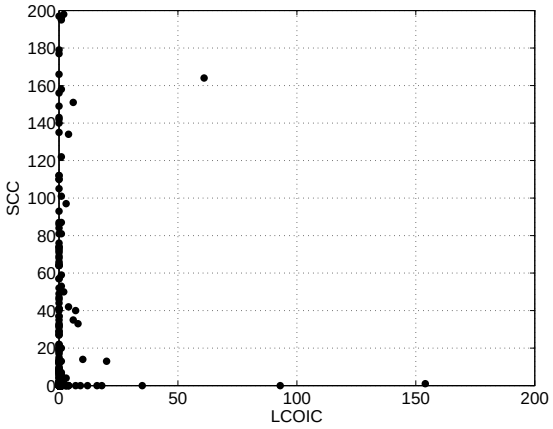


5: Hibernate3

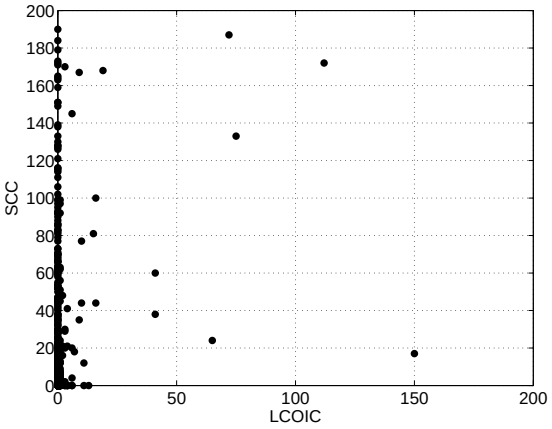


6: JabRef

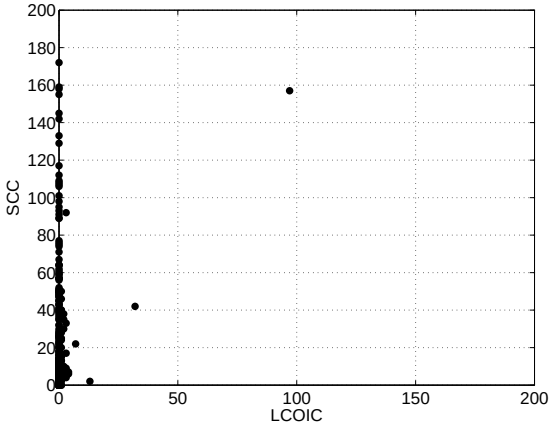
Figure D.9: LCOIC distribution scatter plots



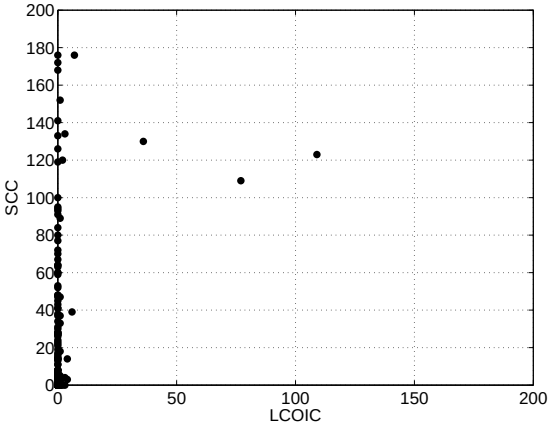
7: JEdit



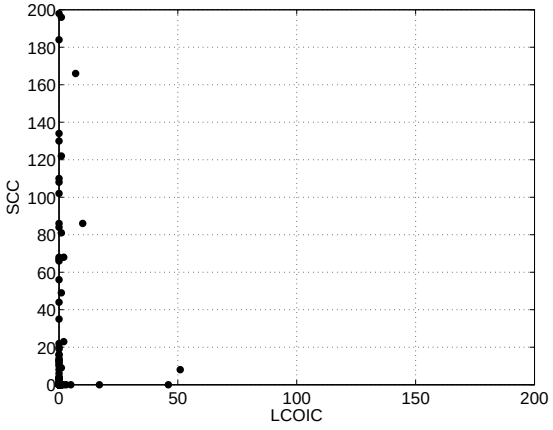
8: Jena2



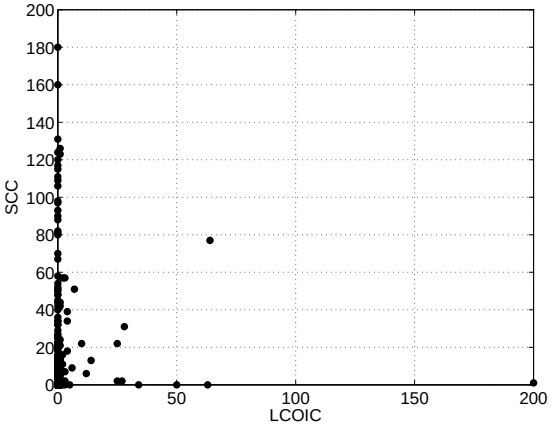
9: JFreeChart



10: Joone

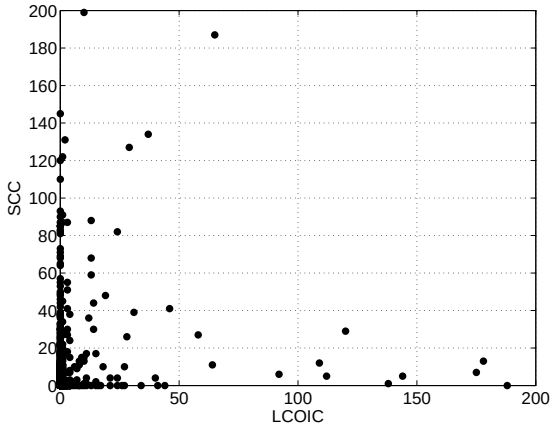


11: Log4j

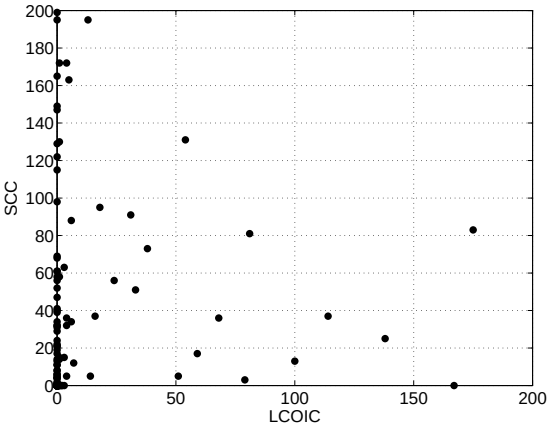


12: Lucene

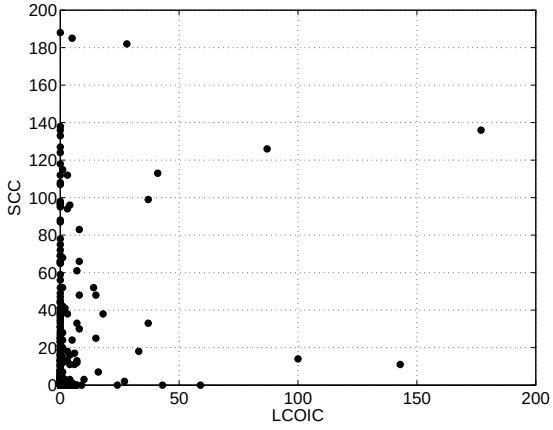
Figure D.9: LCOIC distribution scatter plots (2)



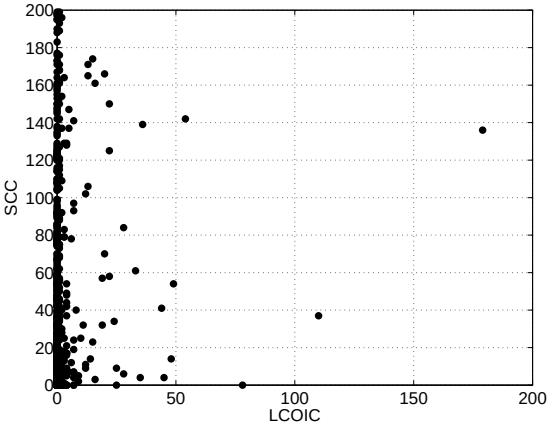
13: RapidMiner



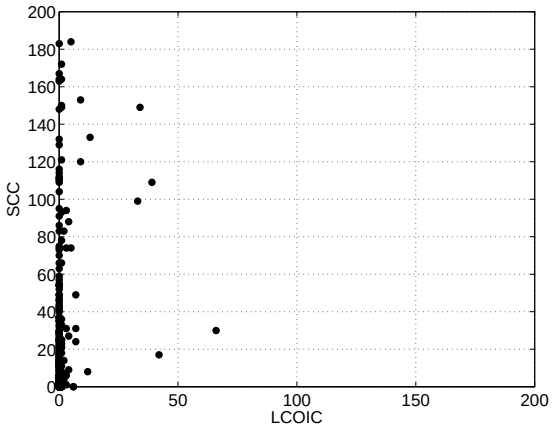
14: Sweet Home 3D



15: TripleA



16: Vuze



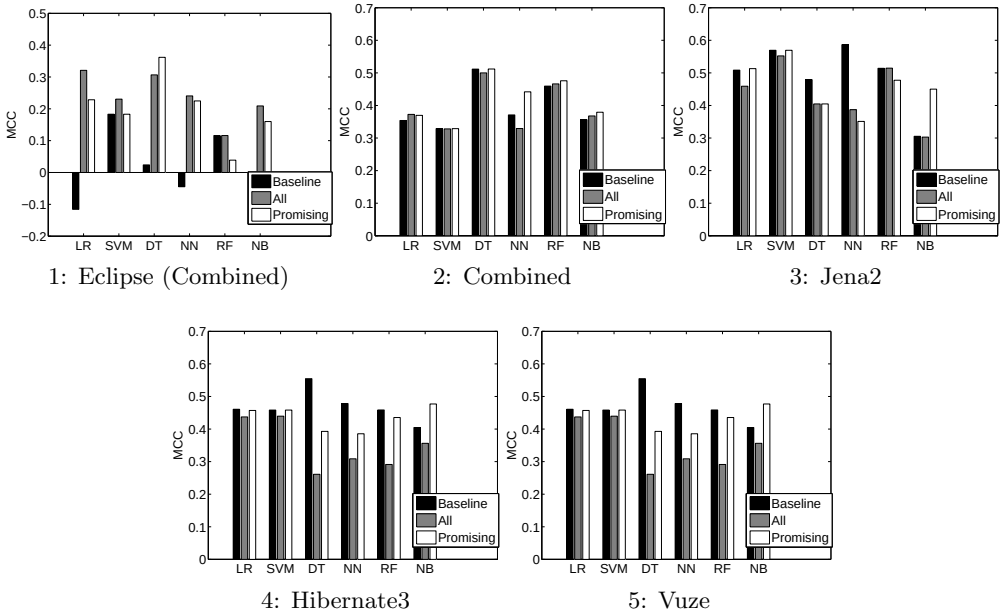
17: Xerces

Figure D.9: LCOIC distribution scatter plots (3)

Appendix E

Detailed Prediction Model Results

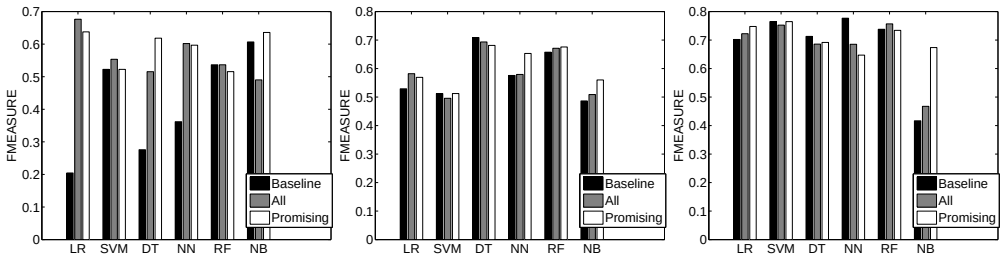
E.1 Core experiments



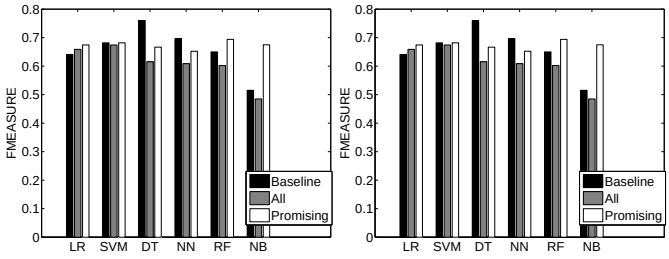
1: Eclipse (Combined) 2: Combined 3: Jena2

4: Hibernate3 5: Vuze

Figure E.1: MCC prediction models core experiment.

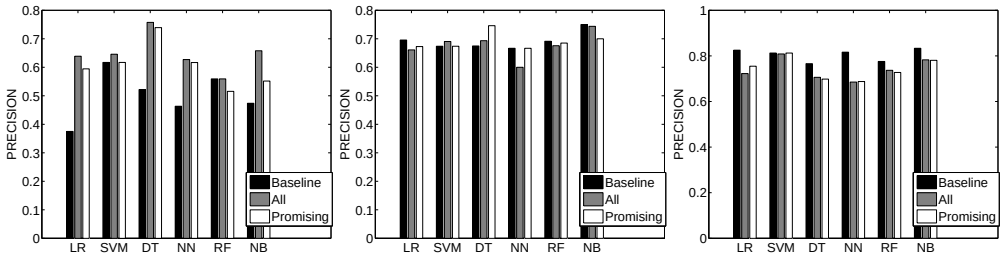


1: Eclipse (Combined) 2: Combined 3: Jena2

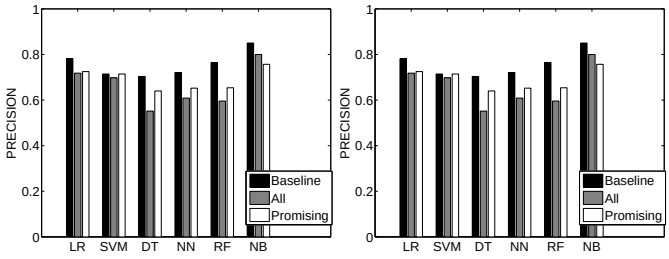


4: Hibernate3 5: Vuze

Figure E.2: F-Measure prediction models core experiment.

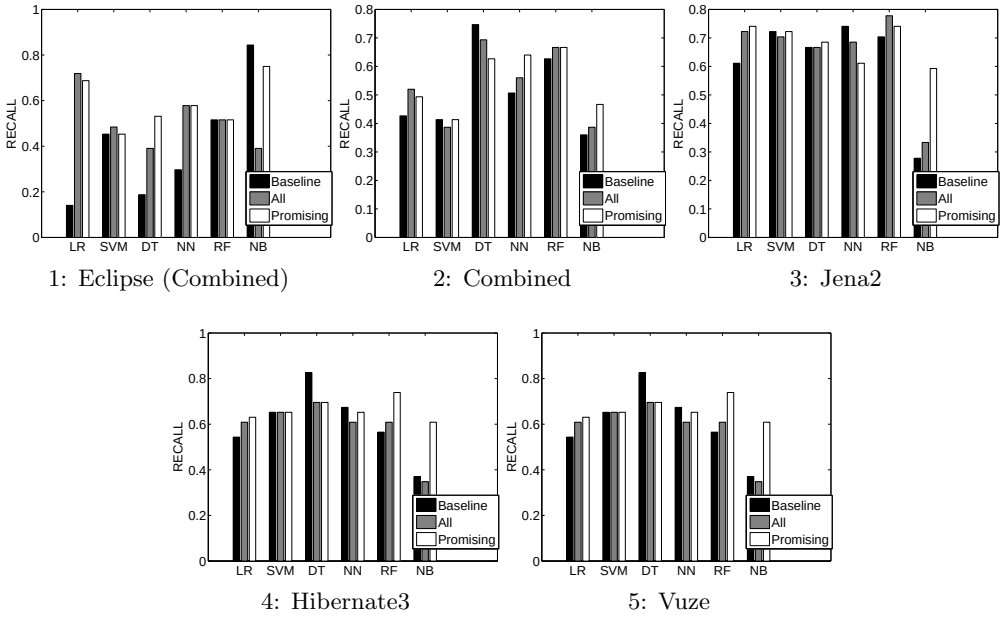


1: Eclipse (Combined) 2: Combined 3: Jena2



4: Hibernate3 5: Vuze

Figure E.3: Precision prediction models core experiment.

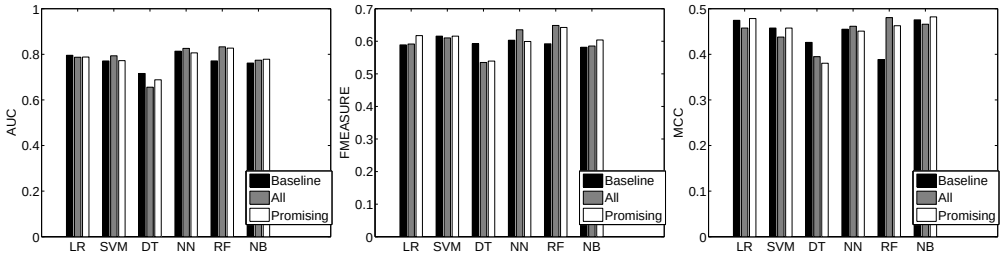


1: Eclipse (Combined) 2: Combined 3: Jena2

4: Hibernate3 5: Vuze

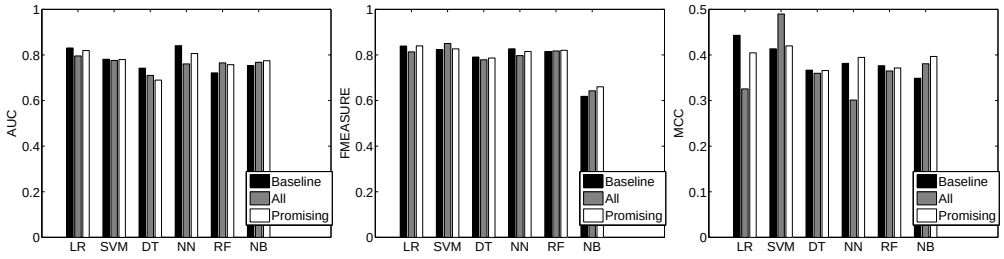
Figure E.4: Recall prediction models core experiment.

E.2 Concrete classes

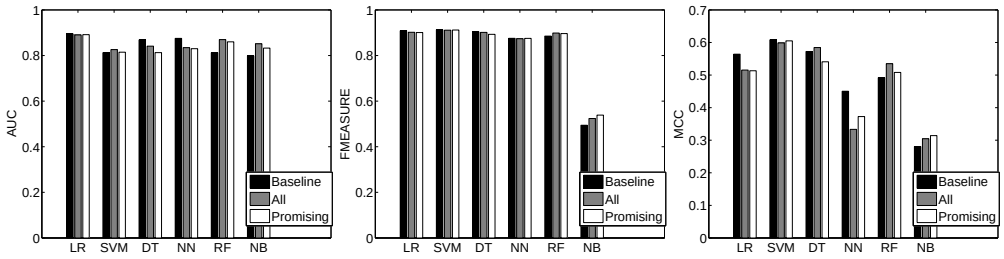


1: ArgoUML-App: AUC 2: ArgoUML-App: F-Measure 3: ArgoUML-App: MCC

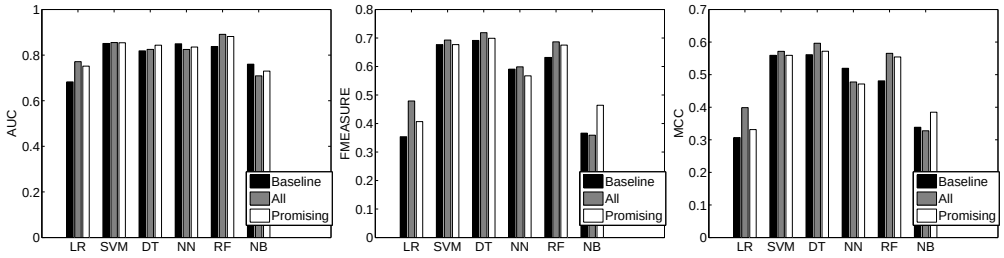
Figure E.5: Prediction model results concrete classes.



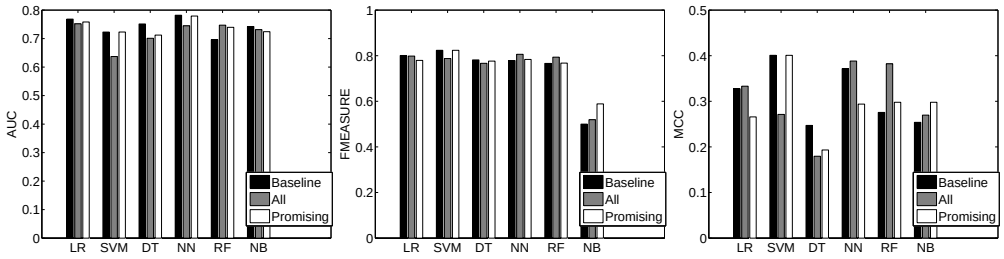
4: Eclipse.debug.ui: AUC 5: Eclipse.debug.ui: F-Measure 6: Eclipse.debug.ui: MCC



7: Eclipse.jdt.core: AUC 8: Eclipse.jdt.core: F-Measure 9: Eclipse.jdt.core: MCC

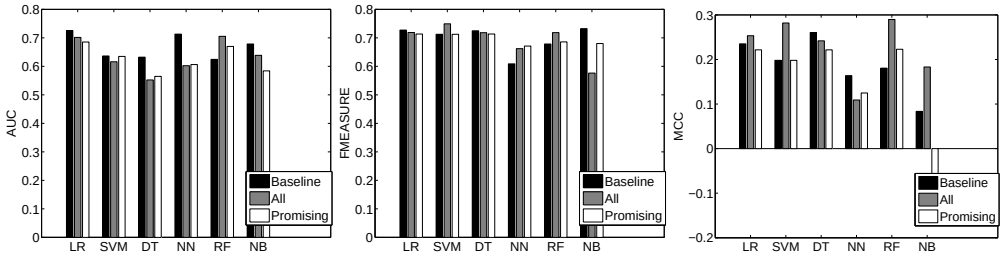


10: Hibernate3: AUC 11: Hibernate3: F-Measure 12: Hibernate3: MCC

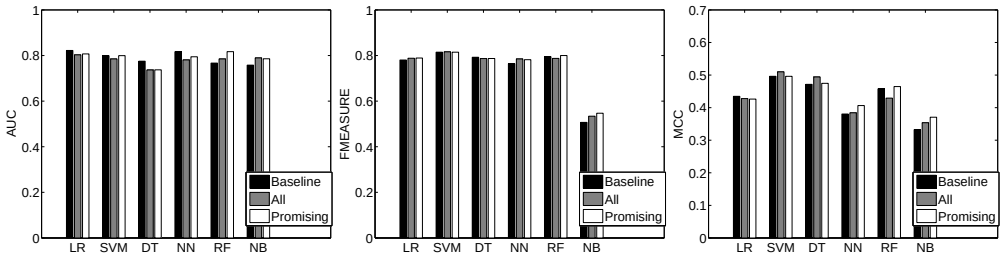


13: JabRef: AUC 14: JabRef: F-Measure 15: JabRef: MCC

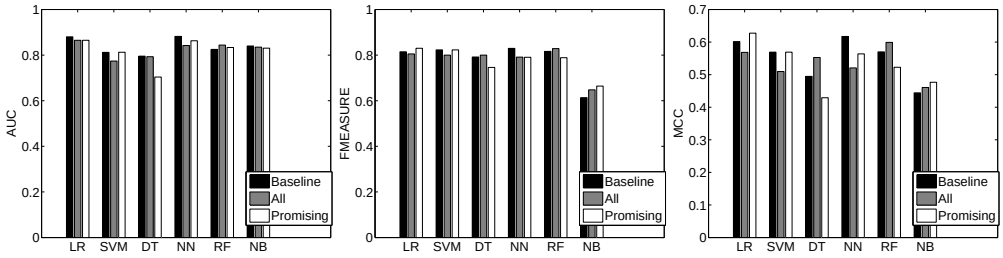
Figure E.5: Prediction model results concrete classes.



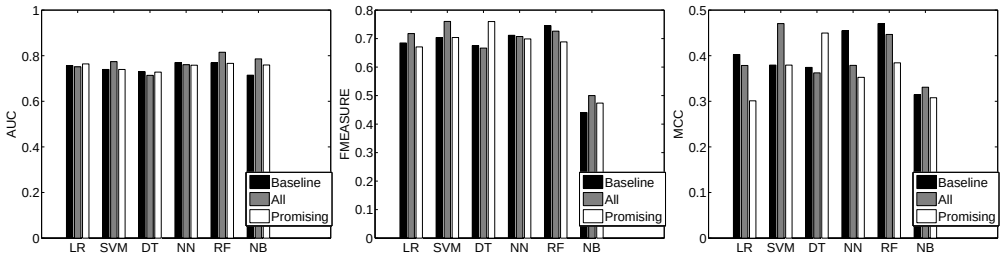
16: JEdit: AUC 17: JEdit: F-Measure 18: JEdit: MCC



19: Jena2: AUC 20: Jena2: F-Measure 21: Jena2: MCC

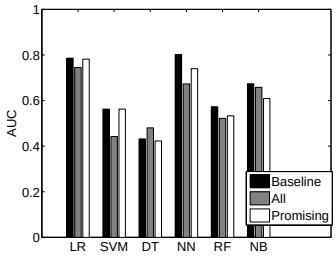


22: JFreeChart: AUC 23: JFreeChart: F-Measure 24: JFreeChart: MCC

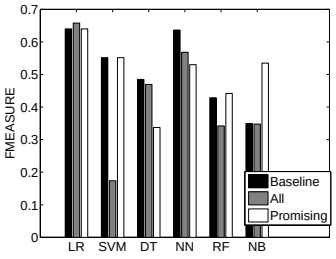


25: Joone: AUC 26: Joone: F-Measure 27: Joone: MCC

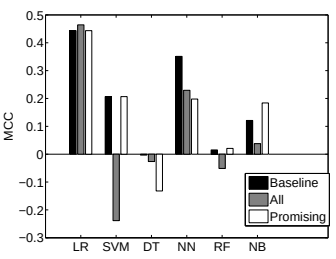
Figure E.5: Prediction model results concrete classes.



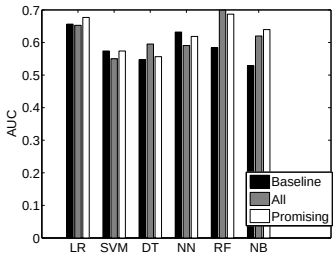
28: Log4J: AUC



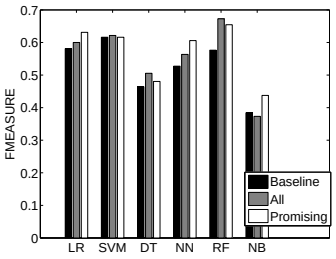
29: Log4J: F-Measure



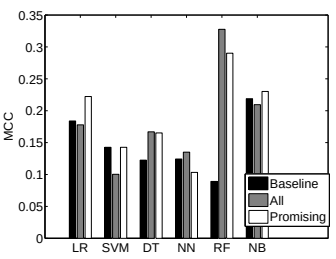
30: Log4J: MCC



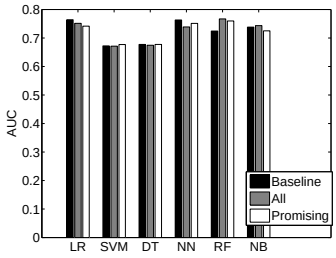
31: Lucene: AUC



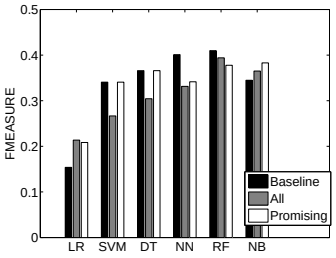
32: Lucene: F-Measure



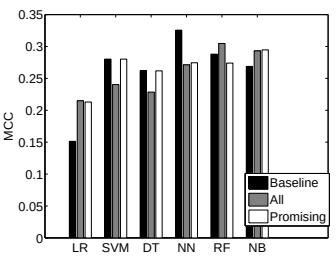
33: Lucene: MCC



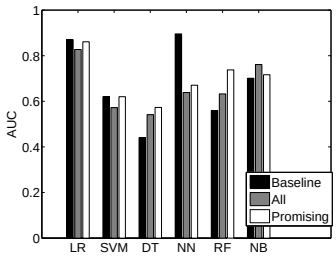
34: RapidMiner: AUC



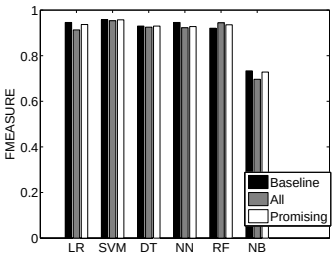
35: RapidMiner: F-Measure



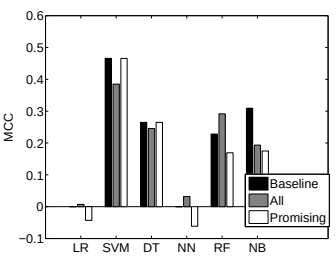
36: RapidMiner: MCC



37: Sweet Home 3D: AUC

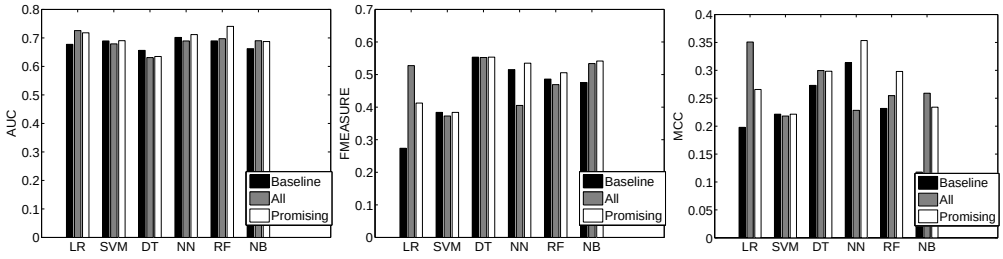


38: Sweet Home 3D: F-Measure

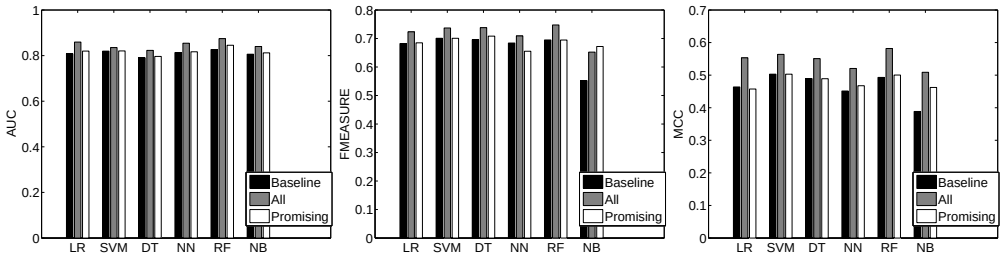


39: Sweet Home 3D: MCC

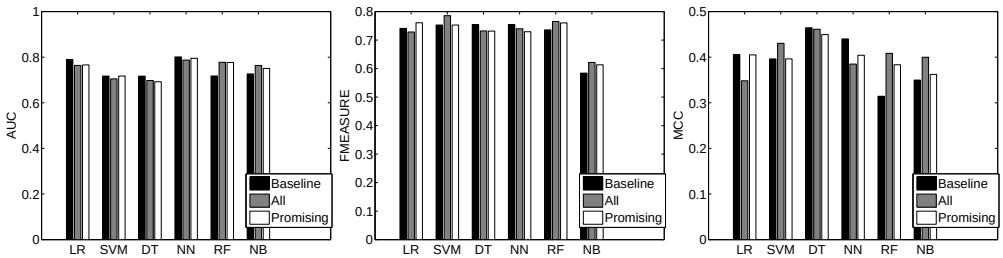
Figure E.5: Prediction model results concrete classes.



40: TripleA: AUC 41: TripleA: F-Measure 42: TripleA: MCC



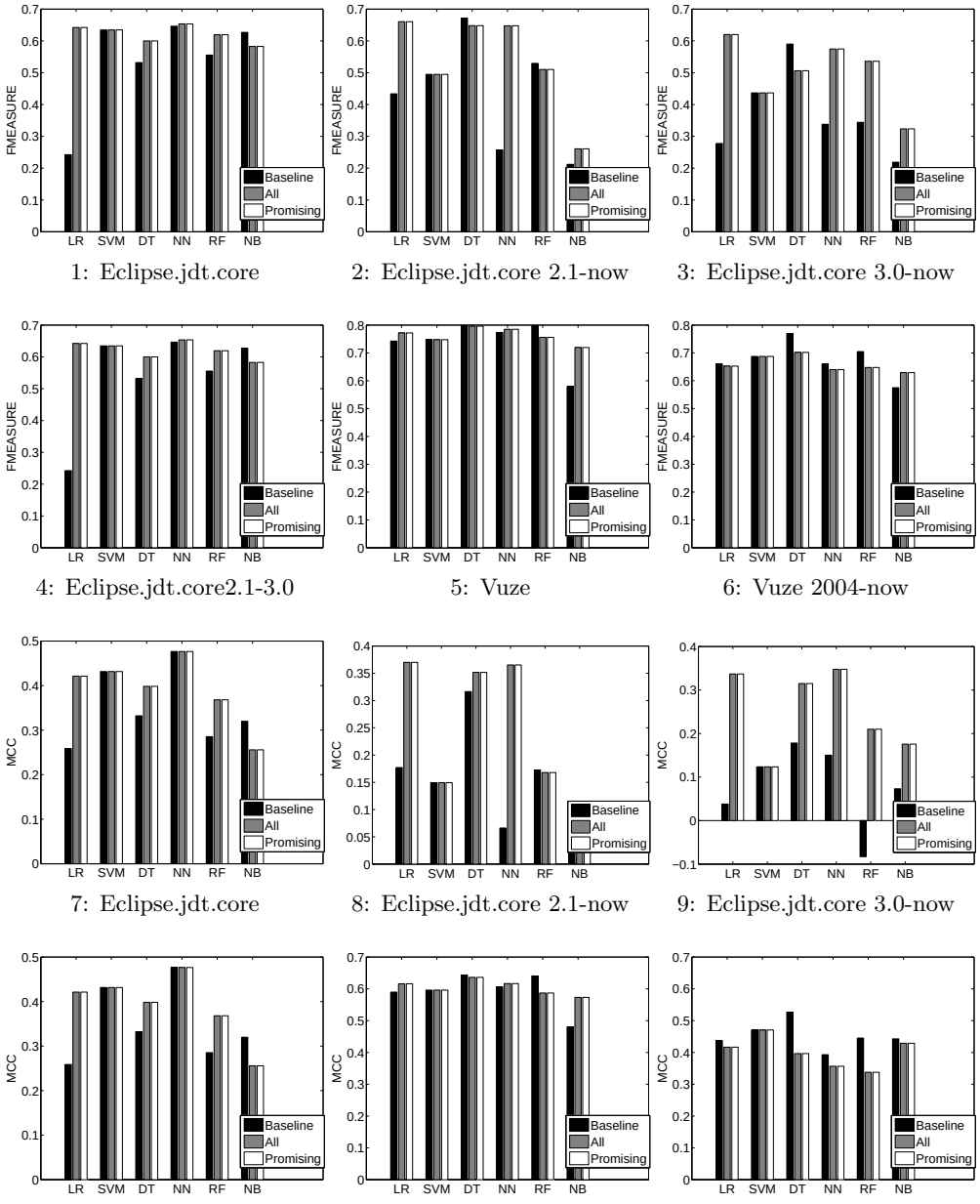
43: Vuze: AUC 44: Vuze: F-Measure 45: Vuze: MCC



46: Xerces: AUC 47: Xerces: F-Measure 48: Xerces: MCC

Figure E.5: Prediction model results concrete classes.

E.3 Prediction models for different releases



1: Eclipse.jdt.core 2: Eclipse.jdt.core 2.1-now 3: Eclipse.jdt.core 3.0-now

4: Eclipse.jdt.core2.1-3.0 5: Vuze 6: Vuze 2004-now

7: Eclipse.jdt.core 8: Eclipse.jdt.core 2.1-now 9: Eclipse.jdt.core 3.0-now

10: Eclipse.jdt.core2.1-3.0 11: Vuze 12: Vuze 2004-now

Figure E.6: Prediction model results for different releases:MCC and F-Measure

E.4 Prediction models for different change-proneness definitions

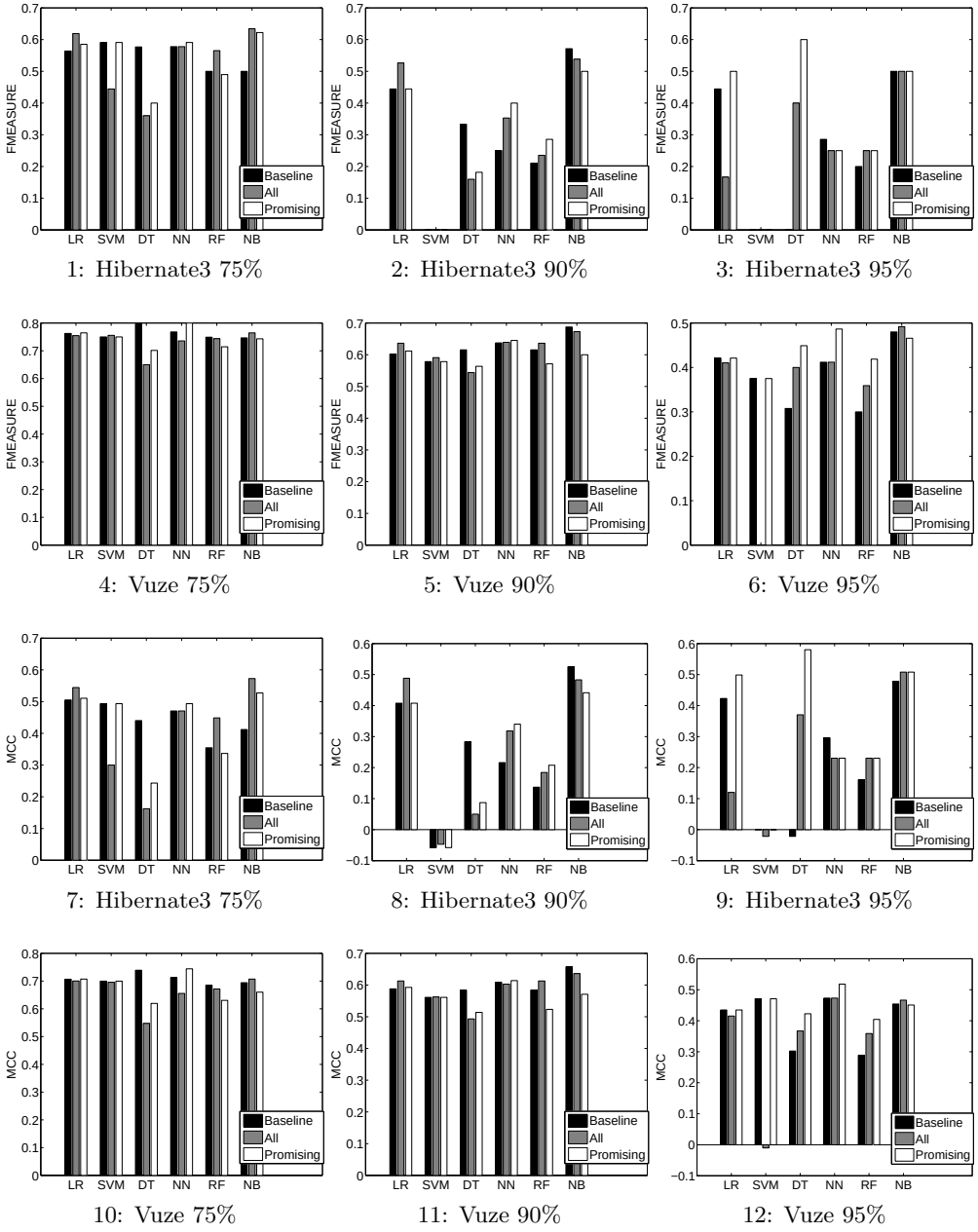


Figure E.7: Prediction model results for different change-proneness definitions:MCC and F-Measure

Appendix F

Custom Versioning History Importers

In this research, we found that the CVS and SVN importers provided in the Evolizer framework suffer from performance issues. As this thesis research requires the versioning histories of many open source projects, the development of faster importers turned out to be worth the investment. Section F.1 describes the custom SVN importer, and Section F.2 describes the custom CVS importer.

F.1 Custom SVN importer

Importing SVN repositories can take up to days or even weeks for large projects. To deal with these problems, we have developed a custom SVN importer. It works as follows:

1. Import all log messages.
2. Checkout the first version, and import all files to the Evolizer database.
3. For each revision, update to that revision and import all file contents.

The main advantage of this approach is the reduction of SVN commands that are sent to the server. The Evolizer SVN importer manually checks out each revision of each file, and thus sends many more commands to the SVN server. Although this does not directly have to be a performance limitation, our initial experiments showed that the custom SVN importer was much faster.

F.1.1 Implementation

Currently, the custom SVN importer is implemented as a standalone Java application using two libraries:

- SVNkit, an open source SVN client implemented in Java.[1]

[1] SVNKit: http://svnkit.com/

- MySQL Connector/J, the MYSQL JDBC driver.[2]

F.1.2 Limitations and future work

The custom SVN importer was developed as a support tool for the research in this thesis, and thus is implemented as an experimental project. Current limitations include:

- Only imports the trunk of the repository, excluding other branches.
- Does not use the Evolizer Hibernate model.
- The SVN importer is not integrated in the Evolizer framework.

For future work, we plan to integrate the custom SVN importer in the Evolizer framework.

F.2 Custom CVS importer

Although the CVS importer in the Evolizer framework is much faster than the SVN importer, there is room for improvement. The Evolizer CVS importer fetches each revision of each file in the repository, requiring a continuous connection with the CVS server during the import process.

Using the backup functionalities of SourceForge[3] and other project repositories, we are able to download a complete CVS history through one command. Then, we can run our own custom importer to import the repository to the database.

F.2.1 Implementation

Currently, the CVS importer is written as a standalone Java application. To store files to the MYSQL database, it uses the MYSQL Connector/J.

F.2.2 Future work

The CVS importer in Evolizer does not suffer from the same limitations as the SVN importer. We think it is possible to use the backup systems to setup a mirror CVS server on a local server. Then, the Evolizer CVS importer can be used to import the repository as usual. An improvement on the current CVS importer could be to add functionalities that automatically create such a mirror.

[2] MYSQL Connector/J: http://dev.mysql.com/downloads/connector/j/
[3] http://www.sourceforge.net