A NEW COUPLING METRIC: COMBINING STRUCTURAL AND SEMANTIC RELATIONSHIPS

A Dissertation
Submitted to the Graduate Faculty
of the
North Dakota State University
of Agriculture and Applied Science

By

Mamdouh Khalaf Alenezi

In Partial Fulfillment of the Requirements
for the Degree of
DOCTOR OF PHILOSOPHY

Major Program:
Software Engineering

June 2014

Fargo, North Dakota

North Dakota State University
Graduate School

Title

A NEW COUPLING METRIC: COMBINING STRUCTURAL AND SEMANTIC RELATIONSHIPS

By

Mamdouh Khalaf Alenezi

The Supervisory Committee certifies that this ***disquisition*** complies with North Dakota State University's regulations and meets the accepted standards for the degree of

DOCTOR OF PHILOSOPHY

SUPERVISORY COMMITTEE:

Dr. Kenneth Magel
Chair

Dr. Rui Dai

Dr. Sameer Abufardeh

Dr. Achintya Bezbaruah

Approved:

6/17/2014
Date

Anne M. Denton
Department Chair

ABSTRACT

Maintaining object-oriented software is problematic and expensive. Earlier research has revealed that complex relationships among object-oriented software entities are key reasons that make maintenance costly. Therefore, measuring the strength of these relationships has become a requirement to develop proficient techniques for software maintenance. Coupling, a measure of the interdependence among software entities, is an important property for which many software metrics have been defined. It is widely agreed that the level of coupling in a software product has consequences for its maintenance.

In order to understand which aspects of coupling affect quality or other external attributes of software, this dissertation introduces a new coupling metric for object-oriented software that combines structural and semantic relationships among methods and classes. The dissertation studies the usage of the new proposed coupling metric throughout change impact analysis, predicting fault-prone and maintainable classes. Three empirical studies were performed to evaluate the new coupling metric and established three results. Firstly, the new coupling metric can be effectively used to specify other classes that might potentially affected by a change to a given class. Secondly, a significant correlation between the new coupling metric and faults has been found. Finally, it has been found that the new metric shows a good promise in predicting maintainable classes. We expect that this new software metric contributes to the improvement of the design of incremental change of software and thus lead to increasing software quality and reducing software maintenance costs.

ACKNOWLEDGMENTS

I sincerely thank Allah the Most Gracious, the Most Merciful for enlightening my mind, making me understand, giving me confidence to pursue my doctoral studies at North Dakota State University, and introducing me to many good people who helped me on my journey.

I would like to express my sincere gratitude and deep appreciation for Professor. Kenneth Magel, my advisor, for his meticulous feedback, exemplary guidance, caring, experience, patience, assistance in every step during my journey in NDSU. He is a real mentor and very inspirational. Throughout my studies, he provided encouragement, sound advice, and lots of good ideas.

I would like to extend my appreciation and thank the esteemed members of my dissertation committee, Dr. Achintya Bezbaruah, Dr. Rui Dai, and Dr. Sameer Abufardeh for generously offering their precious time, support, guidance and good will throughout the preparation and review of this dissertation.

My acknowledgments go to my sponsors: Prince Sultan University and the Ministry of Higher Education in Saudi Arabia represented in the Saudi Cultural Mission and the Royal Embassy in the USA for the full scholarship was given to me. They granted me the scholarship to pursue my PhD at the Department of Computer Science at North Dakota State University. I would also thank the helpful administration and staff at the Department of Computer Science including Stephanie Sculthorp and Carole Huber.

Finally and foremost, my special and deepest appreciations go out to my family members to whom I owe so much. I will forever be deeply indebted to my parents for their

endless love, prayers, and unconditional support not only throughout my doctoral program but also throughout my entire life. Thanks to my brothers and sisters for their continuous support, encouragement and prayers that made me achieve this goal. My heartiest gratitude goes to my lively sons Faisal and Ayoub.

DEDICATION

To my parents and my sons.

TABLE OF CONTENTS

ABSTRACT iii

ACKNOWLEDGMENTS iv

DEDICATION vi

LIST OF TABLES x

LIST OF FIGURES xi

CHAPTER 1. INTRODUCTION 1

1.1. Motivation 4

1.2. Problem Statement and Objectives 4

1.3. Dissertation Contributions 6

1.4. Dissertation Organization 6

1.5. List of Acronyms 7

CHAPTER 2. BACKGROUND AND RELATED WORK 9

2.1. Background 9

2.1.1. Software Metrics 9

2.1.2. Coupling 13

2.1.3. Software Dependencies 18

2.1.4. Latent Semantic Indexing (LSI) 18

2.1.5. Product Metrics in Software Maintenance 25

2.2. Related Work 30

2.2.1. Structural Coupling Metrics 30

2.2.2. Semantic Coupling Metrics 31

CHAPTER 3. COMBINING STRUCTURAL AND SEMANTIC COUPLING . . . 33

3.1. Structural Coupling Relationships 34

3.2. Semantic Coupling Relationships 35

3.3. Combined Coupling Metric 37

3.4. A Toy Example of Measuring the New Coupling Metric 41

3.5. Tool Implementation 43

CHAPTER 4. ASSESSMENT OF THE NEW METRIC 45

4.1. Statistical Evaluation 45

4.1.1. Coupling Metrics 45

4.1.2. Subject Software Systems 47

4.1.3. Measurement Results 48

4.1.4. Cluster Analysis 50

4.2. Analytical Evaluation 51

CHAPTER 5. EMPIRICALLY EVALUATING THE NEW METRIC 56

5.1. Research Questions 56

5.2. Evaluation Methods 57

5.2.1. Accuracy Measures 57

5.2.2. Correlation Analysis 57

5.2.3. Feature Ranking 57

5.3. Impact Analysis 59

5.3.1. Data Collection 59

5.3.2. Empirical Comparison 59
5.4. Fault-Proneness 62
5.4.1. Data Collection 62
5.4.2. Correlation Study 63
5.4.3. Feature Ranking Study 63
5.5. Software Maintainability 64
5.5.1. Data Collection 65
5.5.2. Correlation Study 66
5.5.3. Feature Ranking Study 66
5.6. Discussion 67
5.7. Threats to Validity 68
CHAPTER 6. CONCLUSIONS AND FUTURE WORK 70
6.1. Conclusions 70
6.2. Future Work 71
REFERENCES 73

LIST OF TABLES

Table Page

1. Product Metrics Internal and External Attributes (Adapted from [43]) 11

2. Comparison of Coupling Mechanisms 16

3. Toy Example: Method Dependencies 41

4. Toy Example: Direct Dependencies Between Methods 41

5. Toy Example: The Cosine Similarities Between Methods 42

6. Toy Example: MPC numerator values for method-pairs in C_1 and C_2 42

7. Coupling Metrics 46

8. Selected Software Systems 48

9. Descriptive Statistics for the Coupling Metrics 48

10. Pearson Rank Correlations Between Coupling Metrics 49

11. Details of the Collected History Changes for the Selected Projects 59

12. Precision (P) and recall (R) for impact analysis based on 5 cut points 61

13. Precision (P) and recall (R) for impact analysis based on 10 cut points 61

14. Precision (P) and recall (R) for impact analysis based on 15 cut points 62

15. The results of two Kruskal-Wallis tests for precision and recall 62

16. Spearman Correlation Between Coupling Metrics and Number of Faults 63

17. Ranking of Coupling Metrics Based on ReliefF for Fault-proneness 64

18. Spearman Correlation Between Coupling Metrics and Changed LOC 66

19. Ranking of Coupling Metrics Based on ReliefF for Maintainability 67

LIST OF FIGURES

Figure		Page
1.	Example of Loose Coupling and High Cohesion.	2
2.	Example of Semantic Information Derived from Identifiers and Comments.	5
3.	Categories of Software Metrics.	10
4.	Degrees of Coupling.	13
5.	Eder et al. Structural Coupling Types.	15
6.	Coupling Types.	17
7.	LSI Steps.	19
8.	Prediction Using Historical Faults.	27
9.	An Illustration of Structural Coupling.	35
10.	An Illustration of Semantic Coupling.	36
11.	Architecture of the Combined Coupling Component.	43
12.	Cluster Dendrogram of the Coupling Data.	51

CHAPTER 1. INTRODUCTION

Software is pervading every aspect of life. Software connects people throughout the globe. Software performs mathematics when people shop at grocery stores. Software wakes people up in the morning and makes their commute safer. Almost in every technology, software probably plays some part in it.

In various ways, people lives have become dependent on software and the services it controls. The more critical the service is the more important that the software keeps running without problems. Faults need to be fixed as they happen, performance needs to be monitored and improved if needed, new requirements might arise and thus require new features. In other words, software needs to be maintained.

The software's lifetime might span from days to years. The longer the software is available, the more effort is needed for maintenance. For long-lived software, the maintenance effort might even be greater than the effort of the initial implementation. Software maintenance has the highest cost in a software life-cycle [3].

One of the substantial obstacles to software maintenance is the interconnection (coupling) between software entities [102]. Coupling was described in [15] as "the measure of the strength of association established by a connection from one module to another". One of the substantial principles of software engineering is coupling, which has a high impact on object-oriented software maintenance. A good software design principle is to favor within-class relationships (cohesion) over between-class relationships (coupling), resulting in the often recurring axiom "low coupling/high cohesion". Figure 1 shows an example of low coupling and high cohesion classes. Strong coupling between software entities strengthens the dependency of one entity on the others and grows the possibility that changes in one entity may affect the other entities and have detrimental effects on software maintenance.

Preferably, software classes should show low coupling, since highly coupled classes will be challenging to analyze, modify, or test individually. Understanding the coupling

```
class Report
{
   public bool LoadFromFile(string fileName) { … }
   public bool SaveToFile(string fileName) { … }
}

class Printer
{
   public static int Print(Report report) { … }
}

class Example
{
   static void Main()
   {
      Report myReport = new Report();
      myReport.LoadFromFile("DailyReport.xml");
      Printer.Print(myReport);
   }
}
```

Example of loose coupling. None of the methods depend on the others. The methods rely only on some of the parameters, which are passed to them. These methods can be reused in next projects

```
public final class Math
{
  public static int abs(int i)
    {
      return (i < 0) ? -i : i;
    }
  public static long min(long a, long b)
    {
      return (a < b) ? a : b;
    }
   public static long max(long a, long b)
    {
      return (a > b) ? a : b;
    }
.
.
.
.
.
```

Example of high cohesion. System.Math class performs a single task: it provides mathematical calculations and constants. High cohesion indicates that the class solves only one problem.

Figure 1. Example of Loose Coupling and High Cohesion.

between entities is beneficial for a diversity of software development or maintenance activities, such as assessing software quality [20, 77], predicting fault-proneness [49], supporting change impact analysis [26, 88, 46], change propagation [83], and clone management [45].

The coupling importance has encouraged researchers to capture different aspects of software quality by defining different types of metrics. The majority of the coupling metrics are structural, which statically analyze the source code to capture different relationships between entities (e.g., classes), such as the number of calls between two entities, variable accesses, or inheritance relationships. All these relationships are found in the source code. In the last few years, several alternative coupling aspects have been proposed: dynamic coupling, which takes into account program execution call relationships [8]; semantic coupling, which harnesses relations captured from the source code lexicon [88]; and logical coupling, which uses historical data to identify co-changing artifacts [44]. The motivation behind these three coupling aspects (dynamic, semantic, and logical) is to capture useful relationships between software entities that are not captured by structural coupling.

There have been attempts to combine specific structural metrics with particular semantic metrics with weighting criteria for specific purposes. Bavota [13] combined three metrics (Call-based Dependence between Methods (CDM) [13], Structural Similarity between Methods (SSM) [13], and Conceptual Similarity between Methods (CSM) [70]) with weighting criteria and hierarchical clustering in order to support software refactoring. Bavota et al. [14] proposed an approach that examines the structural (Information-Flow-based Coupling (ICP) [64]) and semantic (Conceptual Coupling Between Classes (CCBC) [88]) coupling associations between the same package classes by finding chains of coupled classes. They use these found chains to construct new packages with greater cohesion than the existing packaging structure. These two studies chose specific coupling metrics and employed empirically calculated weighting criteria to achieve a specific goal. The fact that combining these relationships achieved better results conveys that they can complement each other. In this dissertation, a new unified metric is proposed to combine both structural and semantic relationships of methods and classes, which can be used for several purposes. Structural and semantic relationships are complementary, as shown by previous studies [36, 53, 86]. In addition, a recent study [15] found that structural and semantic couplings match well with developers' idea of coupling.

None of the previous work investigated the effect of combining both structural and semantic coupling at method and class levels on different applications such as software maintainability. In this dissertation, a combination of these two couplings is proposed to complement each other and help in several software maintenance applications.

The remainder of the chapter is organized as follows: section 1.1 presents the motivation for this dissertation; section 1.2 presents the problem statement and objectives of this dissertation; section 1.3 presents the contribution of this dissertation; and section 1.4 presents the organization of this dissertation.

1.1. Motivation

Researchers have studied and analyzed structural and semantic relationships between software entities separately. Relationships between software entities can be seen as twofold: unstructured and structured. Unstructured information means source code comments and identifier names where structured information means source code dependencies. In this dissertation, a new coupling metric is proposed that combines structural and semantic coupling.

These two relationships were chosen for four reasons: both of them reside at the static source code (no need for history of changes or execution traces), the orthogonality between them (they measure completely different relationships), previous studies [36, 53, 86] showed that structural and semantic relationships are complementary, and a recent study [15] which found that structural and semantic couplings match well with developers' idea of coupling. The conjecture is that a combination of structural and semantic coupling relationships would much better explain modules interactions. In particular, a new coupling metric is proposed that finds jointed sets of intensely related methods in different classes by analyzing the method calls and the semantic relatedness between methods.

None of the previous work investigated the effect of combining both structural and semantic coupling on different applications such as impact analysis, fault-proneness, and maintainability. In this dissertation, the new coupling metric is evaluated analytically and statistically. In addition, an empirical evaluation is carried out to evaluate the effect of applying this metric on different software maintenance applications.

1.2. Problem Statement and Objectives

Researchers have recognized that structural and semantic coupling metrics have their limitations in predicting software external quality and maintenance [26, 25, 23, 49, 87, 47]. However, their weaknesses might be mitigated by combining them. Combining information from several sources is a procedure called data fusion [61]. The core idea of data fusion

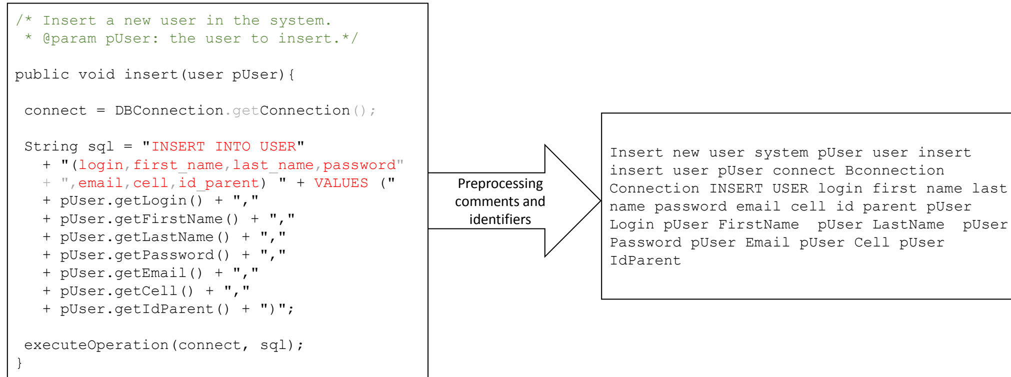

Figure 2. Example of Semantic Information Derived from Identifiers and Comments.

is that to achieve better results, combine information from several sources. This principle has been effectively utilized in different software engineering research areas [35, 109]. The types of data that can be directly retrieved from source code are structural dependencies and semantic information (identifiers and comments). Figure 2 shows an example of semantic information derived from identifiers and comments. This work proposes a new coupling metric that combines structural and semantic sources of information to support better coupling comprehension.

In this dissertation, a preliminary step is taken to answer the research question "what are possibly harmonious profits of combining structural and semantic information for software maintenance applications?". Although both these sources previously have been examined individually, their joint use has not been scientifically investigated. A new metric is proposed that combines both structural and semantic coupling relationships. The hypothesis is that combining both structural and semantic coupling is a good indicator of how well the system is designed and which software element (i.e., classes and methods) is more subject to errors.

The dissertation addresses the following research questions:

- RQ1: Is the new metric (Structural Semantic Coupling Metric (SSCM)) analytically valid?

- RQ2: What are the statistical relationships between the new metric and other coupling metrics?
- RQ3: Does SSCM provide better support for ranking classes during impact analysis than other coupling metrics?
- RQ4: Does SSCM positively correlate with faults, and how much does SSCM contribute to distinguishing faulty and not faulty classes?
- RQ5: Does SSCM positively correlate with class maintainability, and how much does SSCM contribute to identifying maintainable classes?

1.3. Dissertation Contributions

The overall contribution of this dissertation is to combine structural and semantic coupling relationships. This dissertation makes the following contributions:

- Propose a new coupling metric that combines structural and semantic coupling.
- Analytically and statistically evaluate the new coupling metric.
- Perform three different empirical studies to evaluate the new metric in the scope of software maintenance. Specifically, change impact analysis, fault-proneness prediction, and maintainability prediction.

Through the aforementioned contributions my research is expected to guide software maintainers and fill gaps in knowledge of software metrics research: one being a better understanding of the nature of coupling, and the other being insight into the relationship between coupling and maintenance.

1.4. Dissertation Organization

The remainder of this dissertation is organized into six chapters. Chapter 2 starts by introducing some areas that are related to the dissertation and then it discusses related work.

Chapter 3 introduces several coupling relationships and the new coupling metric proposed in this dissertation. Chapter 4 statistically and analytically validates the newly proposed coupling metric. Chapter 5 evaluates the new coupling metrics in three different software maintenance applications. Finally, Chapter 6 presents concluding remarks (including the analysis of the practical applicability of the derived metrics) and outlines some future research directions.

1.5. List of Acronyms

- **LSI**: Latent Semantic Indexing
- **CDM**: Call-based Dependence between Methods
- **SSM**: Structural Similarity between Methods
- **ICP**: Information-Flow-based Coupling
- **CCBC**: Conceptual Coupling Between Classes
- **LOC**: Line of Code
- **KLOC**: Thousands of Lines of Code
- **LCOM**: Lack of Cohesion Of Methods
- **VSM**: Vector Space Model
- **SVD**: Singular Value Decomposition
- **tf-idf**: term frequency-inverse document frequency
- **DIT**: Depth of Inheritance Tree
- **CBO**: Coupling Between Object classes
- **RFC**: Response for a class

- **Ca**: Afferent couplings
- **Ce**: Efferent couplings
- **NPM**: Number of Public Methods
- **IC**: Inheritance Coupling
- **CBM**: Coupling Between Methods
- **CoCC**: Conceptual Coupling of Classes
- **RTC**: Relational Topic-based Coupling
- **LDA**: Latent Dirichlet Allocation
- **RTM**: Relational Topic Models
- **TFC**: Textual Feature Coupling
- **MPC**: Method Pair Coupling
- **HCMC**: Hybrid Coupling between Method and a Class
- **HCCC**: Hybrid Coupling between two classes
- **SSCM**: Structural and Semantic Coupling Metric

CHAPTER 2. BACKGROUND AND RELATED WORK

2.1. Background

This section describes the theoretical context for this dissertation. First, a background on software metrics is provided. Second, a discussion about coupling in object oriented software systems is given. Third, a brief overview of software dependencies is provided. Fourth, Latent Semantic Indexing (LSI) is defined in more detail. Fifth, several applications of product metrics in software maintenance are presented.

2.1.1. Software Metrics

To analyze the state of software metrics, its history needs to be discussed first. Even though the first devoted software metrics book was published in 1977 [48], software metrics history goes back to the 1960s when the lines of code metric (LOC) was used to measure programming productivity and effort. LOC was being used as a measure of program size. In 1971, Akiyama [4] published the first study attempting to employ metrics as software quality predictors in which he used module defect density in terms of the module size measured in KLOC.

In the mid-1970s, observable drawbacks of using LOC as a measure were recognized. The demand for more accurate measures became compelling. A LOC is very different in an assembly language compared to a high-level language. Hence, starting from the mid-1970s, several attempts made to measure software complexity, such as [51, 73] and measures of size, such as [5], which were expected to be programming language independent.

In this section, a survey of the taxonomy of software metrics is presented. There are four categories of software metrics [1] (Figure 3). This classification is based on what they measure and what area of software development they focus on. At a very high level, software metrics can be classified as:

- Process metrics: They coincide with the software development process, encompassing the standards, activities, and methods used. They leverage historical project

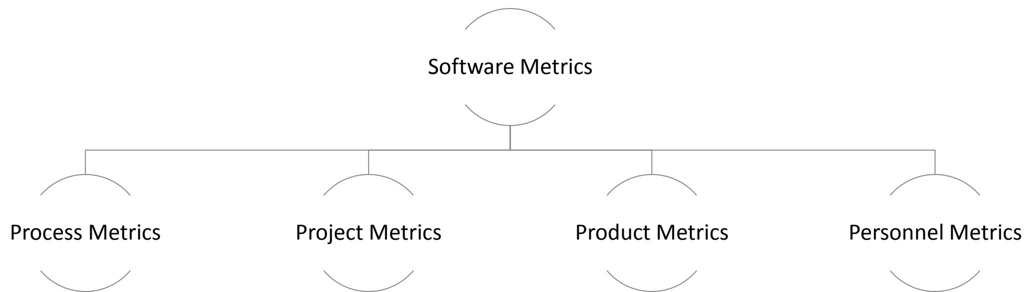

Figure 3. Categories of Software Metrics.

knowledge and asses the capability of the software engineering process. Examples of process metrics are prior defects, prior commits, and metrics for measuring project progress. For example, efficiency of fault detection.

- Project metrics: They indicate how the project is planned and executed. The intent of these metrics is twofold: minimize development schedule and assess product quality. Examples include number of developers, effort allocated per phase, and amount of design reuse achieved by the project.

- Product metrics: They measure characteristics of the result of a software development process. Product metrics are typically calculated from source code. Product metrics refer to different features of the product such as design features, size, complexity, and performance. They provide software engineers with a guide to analyze, design, code, and test software more objectively. Examples include complexity of the design, size of the source code, and usability of the produced documentation.

- Personnel metrics: Personnel metrics indicate the productivity and quality for each of the project team members. Examples of personnel metrics are programming experience, communication level, and work load. For example, ranking developers based on their programming experience (low, moderate, high).

Product metrics can be categorized as internal and external attributes. Internal product attributes measure the product itself. Internal attributes are concerned with size, com-

plexity, coupling, and cohesion. External attributes are concerned with product quality, such as usability, testability, reusability, and portability [42]. As external attributes are directly observable only after the system has already been deployed and operational for some time, the focus has been on relating internal attributes (software metrics) to their external qualities. Table 1 presents several examples of internal and external attributes for product metrics.

Table 1. Product Metrics Internal and External Attributes (Adapted from [43])

Product	**Attributes**	
	Internal	External
Specification	size, reusability, modularity, redundancy, functionality, correctness	comprehensibility, maintainability, volatility
Design	size, reuse, modularity, coupling, cohesiveness, functionality	quality, complexity, maintainability
Code	size, reuse, modularity, coupling, functionality, algorithmic complexity, control-flow, structuredness	reliability, usability, maintainability

In the following paragraphs, a description of product internal attributes is presented, including size, complexity, coupling, and cohesion. To measure source code size counting the number of lines of code (LOC) is the easiest way . However, this metric has its flaws. For example, it is possible to write the same functionality with fewer (or more) lines of code, while sustaining the same complexity. The LOC metric has been utilized for several purposes, such as measuring size and assessing programming effort.

Complexity is a measure of how complex the source code is. Complexity has a negative effect on modifiability and maintainability: lower is better. There are different aspects of structural complexity, and each plays a different role. According to [43] structure has two parts: control flow structure and data flow structure. Representative Examples of complexity-based code-level metrics are Halstead Volume [51], a metric based on operator and operand counts (data flow), and McCabe Complexity [73], a metric based on the number of possible paths in the program control graph (control flow). McCabe's cyclomatic

complexity and its variations capture different flavors of code complexity. Coupling also can be seen from data flow and control flow perspectives. Control coupling is one module controlling the flow of another whereas data coupling is when modules share data through, for example, parameters.

Coupling was defined according to the IEEE standards [55] as ”a measure of the interdependence among modules in a computer program”. Coupling exist between two entities if one accesses or uses some elements of the other entitiy. The assumption is that lower coupling is better. Examples of coupling metrics are CBO, RFC [28], Ca, Ce [72], NPM [11], IC, CBM [100]. Since this dissertation is introducing a new coupling metric, coupling is discussed in more details in section 2.1.2.

Cohesion is defined according to the IEEE standards [55] as ”(1) the manner and degree to which the tasks performed by a single class module are related to one another or (2) in software design, a measure of the strength of association of the elements within a module”. Cohesion quantifies how strongly the responsibilities of a class are related. The rationalization behind measuring cohesion is the belief that classes should emphasis just one abstraction which will improve maintainability. Lack of Cohesion Of Methods (LCOM), one of the most common cohesion metrics, implies that the class should be split into two or more sub classes. Non-cohesive classes are difficult to reuse. Examples of cohesion metrics are LCOM1, LCOM2 [28], LCOM3, LCOM4 [54], LCC, and TCC.

Product metrics have come to be important in a subset of software engineering disciplines, software quality and maintenance. They are used to estimate the effort and cost of software projects and to measure software quality [42]. In software evolution, product metrics are utilized for recognizing the stability of software system entities, along with recognizing where refactorings can be or have been practiced, and identifying the variation of quality in the evolving software systems structure. In reverse engineering, product metrics are utilized for evaluating the complexity and quality of software products.

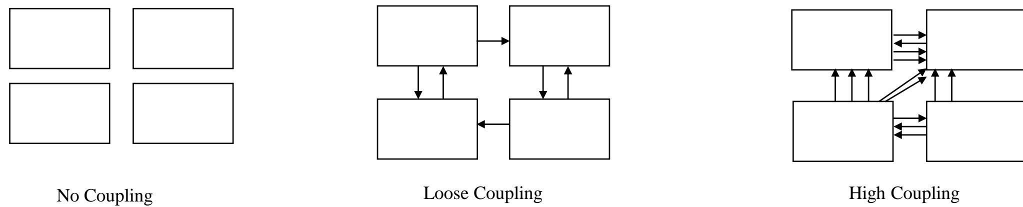

Figure 4. Degrees of Coupling.

To support metrics usefulness in practical applications, it is necessary to empirically validate them. Empirical studies frequently rely on data sets. Since open source software projects are often developed by volunteers with no formalized development methods [49], their process and resource data are often incomplete and unreliable. Consequently, to conduct and validate the empirical work of open source projects, product metrics only are exploited.

2.1.2. Coupling

Coupling is a principle in programming that indicates that a single entity is dependent (or coupled) upon another entity. Coupling is described in [15] as "the measure of the strength of association established by a connection from one module to another". Coupling is an important software internal quality attribute, with a great influence on object-oriented software comprehension. Low coupling and high cohesion are considered as indicators of a well-structured software system. Preferably, software entities should exercise low coupling, since highly coupled entities will be difficult to examine, comprehend, alter, or test individually (Figure 4). Studies have shown that coupling is related to the external quality of a software system, fault-proneness [49], change propagation [83], and change impact analysis [111].

Another definition of coupling was introduced by Wand and Weber: two entities are coupled if and only if the change history of one entity depends on the change history of the other [106]. Various situations may cause two software artifacts to have dependent change histories, (e.g., one controls the functionality of the other; or one uses the data defined by

the other; or one contains code cloned from the other). By definition, all these cases are examples of coupling, although the causes are different.

Myers [74] refined the notion of coupling by classifying coupling into six different levels. However, his definition of coupling was neither accurate nor prescriptive. Constantine and Yourdon [113] detailed that software design modularity could be judged by evaluating cohesion and coupling. They indicated that coupling reveals the interdependencies between entities. Errors in one entity could propagate to the other coupled entity.

Fenton and Melton [41] proposed a theory of software measurement, which offers a basis for specifying software complexity. They included the number of connections between entities into the coupling measure. Traditionally, entity coupling was utilized as an inexact measure of the complexity of software. Offutt and et. al. [75] extended previous work to develop a universal software metric system to measure coupling automatically. They introduced clear definitions of the coupling levels, which can be measured algorithmically, introduced the concept of direction into the coupling levels, and incorporated different types of non-local variables as found in newer programming languages. Chidamber and Kemerer [28] defined six new source code metrics for object-oriented systems and analytically evaluated these new metrics.

Subramanyam and Krishnan [98] argued that the amount of coupling between entities is proportional to the challenge of changing, understanding, and correcting these entities. Alternatively, a strong coupling between classes can increase the complexity of the software system and hence may affect external quality such as reliability and maintainability [58]. Low coupling has been considered as an important characteristic of good software systems because it allows individual modules in a system to be easily modified with relatively little worry about affecting other modules in a system [22].

Several object-oriented coupling frameworks were proposed in the literature. The following paragraphs summarize them. Eder et al. [37] defined three types of coupling

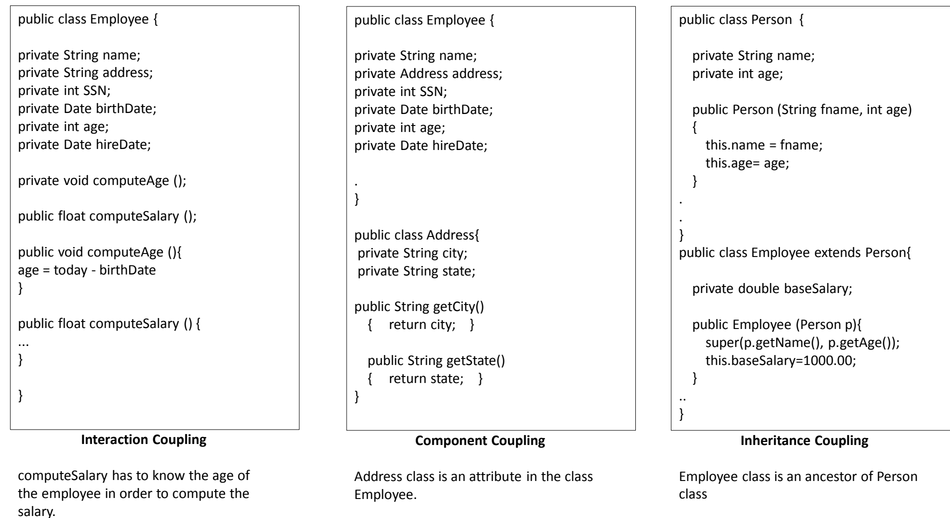

Figure 5. Eder et al. Structural Coupling Types.

namely interaction coupling, component coupling, and inheritance coupling. Figure 5 shows examples of these three coupling types.

1. Interaction coupling: Two methods are coupled, if (i) one method invokes the other, or (ii) they communicate via the sharing of data.

2. Component coupling: Two classes c and d are coupled, if d is either (i) an attribute of c, or (ii) an input or output parameter of a method of c, or (iii) a local variable of a method of c, or (iv) an input or output parameter of a method invoked within a method of c.

3. Inheritance coupling: two classes c and d are coupled, if one class is an ancestor of the other.

Hitz and Montazeri [54] identified two levels of coupling, class level and object level. The class level coupling refers to design-time static dependencies between classes. The object level coupling refers to run-time dynamic dependencies between classes. Class level coupling is important since changes in one class may affect other classes that use it.

Briand et al. [22] attempted to unify coupling metrics for OO systems in a comprehensive framework. They defined coupling metrics by:

- Type of coupling: the mechanism that establishes coupling between two classes, such as method invocation, attribute reference, type of attribute, type of parameters, passing of pointer to method.
- Locus of impact: the method and attribute are used or use other classes or attributes
- Granularity: level of detail where information is collected, such as method, class, module or system level.
- Counting direct or indirect connection
- Inheritance-based or non-inheritance based coupling
- Polymorphism based or non-polymorphism based coupling

Table 2 shows a comparisons between three different coupling frameworks as to what constitutes coupling.

Table 2. Comparison of Coupling Mechanisms

Mechanism	Eder et al.	Hitz	Briand et al.
Methods share data	✓		
Methods reference attribute		✓	
Method invokes method	✓	✓	✓
Class is type of a class' attribute	✓	✓	✓
Class is type of a method's parameter or return type	✓	✓	✓
Class is type of a method's local variable	✓	✓	
Class is ancestor of another class	✓	✓	

Coupling, in general, can be classified, based on how it may be detected, into structural coupling, dynamic coupling [8], evolutionary/logical coupling [44], and semantic coupling [87]. These different couplings model different types of dependencies within a software system (Figure 6).

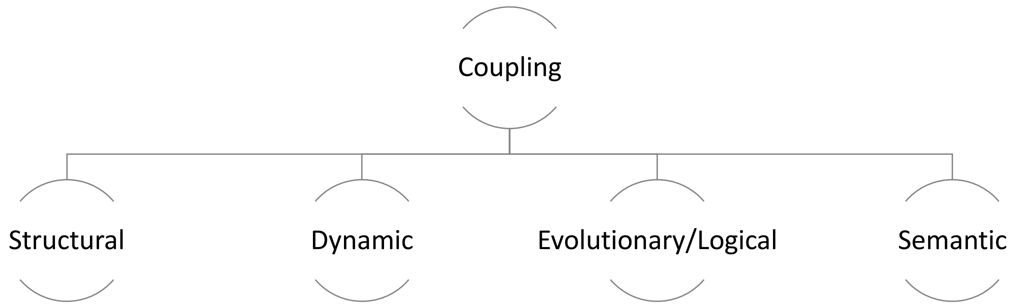

Figure 6. Coupling Types.

- Structural coupling: The most commonly examined type of coupling is structural coupling. It models static compile-time dependencies between source code entities (i.e., method invocations or variable references). Since the compiler must be aware of the dependencies, structural coupling is easy to detect.

- Dynamic coupling: It models run-time interactions between program components observed during an execution of the program. In case of polymorphism or dynamic dispatch, structural coupling analysis sometimes cannot determine which function definition will be called in an object-oriented system. Dynamic profiling can determine these cases by observing which function bodies actually are invoked at run-time [8].

- Evolutionary/logical coupling: Logical coupling is detected from the version control system histories based on observed co-change relations [44]. The idea is that if two artifacts are often changed together in change commits, then they are coupled logically, even if there is no obvious static dependence.

- Semantic coupling: Semantic coupling is based on semantic information extracted from the text within the software systems, such as in comments, identifiers, and meta-data [87]. Semantic coupling measures the strength of the semantic similarity of software artifacts through information retrieval techniques.

2.1.3. Software Dependencies

A basic definition of software dependency was introduced by Parnas [81] as a "uses" relationship. A software entity A uses, or depends on, B when A requires the presence of a correct version of B. Dependency is a connection between two items in which changes to one item may cause changes to the other item. Two kinds of software dependencies can be found in practice: static (compile time) dependencies; and dynamic (run time) dependencies. Static dependencies capture the notion that one software module must be present compile another. Dynamic dependencies, in contrast, are based on actual calling patterns of the software during operation.

In OO programming, dependencies can be defined in terms of certain relationships between software entities. These relationships can be classified into method call, class access, or class inheritance. These relationships are described as follows:

1. Method call: a method call dependency that goes from $Class_1$ to $Class_2$ if there is at least one method within $Class_1$ that invokes at least one method of $Class_2$.

2. Class access: a class access dependency that goes from $Class_1$ to $Class_2$ if $Class_2$ is explicitly used in $Class_1$ code (e.g., $Class_2$ is used within $Class_1$ as a type of an instance and/or a class variable).

3. Class inheritance: a class inheritance dependency that exits $Class_1$ and points to $Class_2$ if $Class_1$ is a subclass of $Class_2$.

2.1.4. Latent Semantic Indexing (LSI)

Latent Semantic Indexing (LSI) is a statistical corpus-based technique used to induce and represent the natural language characteristics and meanings reflective in these natural language characteristics usage [33, 71]. LSI is based on the vector space model (VSM) [16], in which it produces real numbers to describe documents. Several studies showed that LSI encapsulates significant parts of the whole passages meaning such as essays,

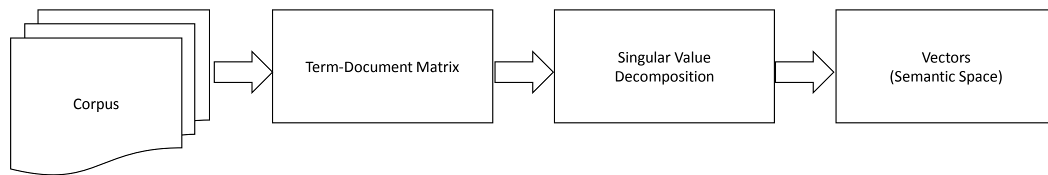

Figure 7. LSI Steps.

paragraphs, and sentences. The LSI basic principle is that the similarity of meaning of a set of words depends on the word context (where it appears). [88].

Usually, the word by context matrix is very large and sparse. LSI uses dimension reduction to capture the essential semantic information using singular value decomposition (SVD) [16]. SVD is a mechanism to reduce the dimensionality and determine patterns in the relationships between the words and concepts used in the documents. Overall, SVD is applied to the term-by-document matrix to construct an LSI subspace.

LSI offers a way to measure, in an unsupervised automatic way, the semantic similarity between any two samples of text. Figure 7 shows the LSI steps. LSI does well in the domain of source code because many informal concepts of the problem are embodied in the names of variables that assigned by the programmers in source code implementation [71]. LSI has been widely used for several software engineering applications such as feature location [30, 92], traceability link recovery [56], software measurement [70, 88], and detecting code clones [99].

The cosine between two documents vectors is usually utilized in text mining tasks to measure the documents similarity. The main feature of LSI is its ability to excerpt the semantic meaning of a text by establishing links between those terms that occur in similar contexts.

The initial step is to build the corpus for the software system. The corpus consists of a set of documents. In this dissertation, documents in the corpus are methods. These documents include the text of each method including all the identifier names, comments,

etc. The preprocessing steps are disregard language keywords, split identifiers, and remove stop words [87]. The extracted source code is processed as follows:

1. Predefined tokens are removed (such as, numbers, operators, special symbols, Java programming language keywords, etc.)

2. The identifier names in the source code are split based on known coding standards while the original form of every identifier is kept too [69, 86]. For instance, these identifiers are broken into the words “first” and “name”: “first_name”, “first_Name”, “firstName”, etc.

3. Stop Words are words which do not contain important significance. Stop words are words which are filtered out prior to processing text. Examples include a, an, about, the, etc.

4. Each document in the corpus is created with the comments and identifiers corresponding to each method in that software system.

Here is an LSI example from source code. Assume that there is three methods in the source code that going to be represented in LSI.

Method A:

```
public Employee (String name, double salary)
{
    this.name= name;
    this.salary= salary;
}
```

Method B:

```
public void increaseSalary(double amount)
{
```

```
    salary += amount;
}
```

Method C:

```
public static void create(Employee emp)
{
    emp = new Employee();
}
```

Each method is going to be represented as a document. The preprocessing steps include removing language keywords and stop words and splitting identifiers. The resulting documents after preprocessing are:

- Method A: “employee name salary name name salary salary”.
- Method B: “increasesalary amount salary amount”.
- Method C: “create employee emp emp employee”.

The first step of LSI is to construct a term-document matrix inwhich each word is represented by a row in the matrix and each document is represented as a column in the matrix. A cell gives the number of times that word appears in that document. The term-document matrix of this example would look like:

$$\begin{bmatrix} Word & MethodA & MethodB & MethodC \\ employee & 1 & 0 & 2 \\ name & 3 & 0 & 0 \\ salary & 3 & 1 & 0 \\ increasesalary & 0 & 1 & 0 \\ amount & 0 & 2 & 0 \\ create & 0 & 0 & 1 \\ emp & 0 & 0 & 2 \end{bmatrix}$$

After constructing the term-document matrix, the values of the matrix are weighted using term frequency-inverse document frequency (tf-idf). tf-idf evaluates the importance of a particular word to a given document. The weight is given as:

$$w_d = f_{w,d} \times log(\frac{M}{f_{w,M}})$$

where $f_{w,d}$ equals the number of times the word (w) appears in a document (d), M is the size of the corpus, and $f_{w,M}$ equals the number of documents in which w appears in M.

The term-document matrix after tf-idf weighting would look like:

$$\begin{bmatrix} 0.176 & 0 & 0.352 \\ 1.431 & 0 & 0 \\ 0.528 & 0.176 & 0 \\ 0 & 0.477 & 0 \\ 0 & 0.945 & 0 \\ 0 & 0 & 0.477 \\ 0 & 0 & 0.945 \end{bmatrix}$$

The next step is to use singular value decomposition (SVD) to convert the term-

document matrix to three matrices. M is an $r \times c$ matrix (r rows and c columns), and T, S, and D^T are the three new matrices.

$$M = T \times S \times D^T$$

where the columns of T are orthonormal eigenvectors of $M \times M^T$, the the columns of D are orthonormal eigenvectors of $M^T \times M$, S is a diagonal matrix with non-negative and decreasing order diagonal elements. The main diagonal elements of S are singular values of M and are the square roots of the eigenvalues of $M^T \times M$ and $M \times M^T$. T and D are matrices whose columns are left and right singular vectors of M where each row in the D matrix represents a document vector.

$$\begin{bmatrix} 0.126 & 0.307 & 0.006 \\ 0.926 & -0.064 & -0.107 \\ 0.350 & -0.034 & 0.123 \\ 0.024 & -0.027 & 0.440 \\ 0.048 & -.055 & 0.881 \\ 0.017 & 0.427 & 0.026 \\ 0.034 & 0.845 & 0.052 \end{bmatrix} T$$

$$\begin{bmatrix} 1.539 & 0 & 0 \\ 0 & 1.114 & 0 \\ 0 & 0 & 1.078 \end{bmatrix} S$$

$$\begin{bmatrix} 0.996 & 0.077 & 0.055 \\ -0.050 & -0.064 & 0.997 \\ -0.08 & 0.995 & 0.060 \end{bmatrix} D^T$$

$$\begin{bmatrix} 0.996 & -0.050 & -0.080 \\ 0.077 & -0.064 & 0.995 \\ 0.055 & 0.997 & 0.060 \end{bmatrix} D$$

Next, SVD reduces the high dimensional documents vectors to low dimensional space by using the low rank approximation. The matrices, after reducing the dimensionality, would look like:

$$\begin{bmatrix} 0.126 & 0307 \\ 0.926 & -0.064 \\ 0.350 & -0.034 \\ 0.024 & -0.027 \\ 0.048 & -0.055 \\ 0.017 & 0.427 \\ 0.034 & 0.845 \end{bmatrix} T_2$$

$$\begin{bmatrix} 1.539 & 0 \\ 0 & 1.114 \end{bmatrix} S_2$$

$$\begin{bmatrix} 0.966 & -0.050 \\ 0.077 & -0.064 \\ 0.055 & 0.997 \end{bmatrix} D_2$$

After representing each document in a reduced form using SVD. The similarity between two documents can be measured using the cosine between them. The cosine can be computed as inner product between document vectors as follows:

$$CosineSimilarity(d_1, d_2) = \frac{d_1 \times d_2}{|d_1| \times |d_2|}$$

$$CosineSimilarity(M_A, M_B) = \frac{(0.996)(0.077) + (-0.050)(-0.064)}{\sqrt{(0.996)^2 + (-0.050)^2}\sqrt{(0.077)^2 + (-0.064)^2}} = 0.8$$

$$CosineSimilarity(M_A, M_C) = \frac{(0.996)(0.055) + (-0.050)(0.997)}{\sqrt{(0.996)^2 + (-0.050)^2}\sqrt{(0.055)^2 + (0.997)^2}} = 0.005$$

$$CosineSimilarity(M_B, M_C) = \frac{(0.077)(0.055) + (-0.064)(0.997)}{\sqrt{(0.077)^2 + (-0.064)^2}\sqrt{(0.055)^2 + (0.997)^2}} = -0.6$$

These similarity results show that there is a similarity between methods ($M_A and M_B$). However there is no similarity between methods ($M_A and M_C$) and ($M_B and M_C$).

2.1.5. Product Metrics in Software Maintenance

Product metrics have been widely used to identify parts of a system that need more attention in maintenance [98, 110, 49]. The use of product metrics within object-oriented systems has been an active area of software engineering research. Subramanyam and Krishnan [98] introduced a short survey of research activity in this area. Several studies attempted to characterize the systems under analysis and relate metrics to quality and maintainability [110]. Product metrics can help the decision-making processes in the software maintenance phase. Particularly, these metrics are simple, easy, and quick to use while positively affecting software maintenance decision-making.

Software maintenance has the highest cost in a software life-cycle. Consequently, any part of maintenance that can be automated will eventually lead to saving maintenance resources. The use of product metrics in different software maintenance applications (change impact analysis, fault-proneness, and maintainability) is presented next.

Software systems undertake regular changes throughout the software maintenance phase. These changes may bring new functionality to the program, fix existing faults, or adapt the system to changes in its environment. While these types of changes are beneficial, they also introduce a risk that the changes have unintended impacts that may cause

the software to behave in an undesirable way. These undesirable behaviors may include injecting a new defect, breaking existing functionality, or decreasing the performance of the application. Due to these risks, it is vital to comprehend the potential impact of a software change as early as possible.

However, change impact is often difficult to trace. In practice, the most frequently used technique for change impact analysis is code inspection. Developers may get some support for this task from their development environment, but the majority of the task is manual which makes the amount of time required to determine the overall impact of a change large and error-prone [2].

An essential step when modifying or maintaining software is impact analysis [90], which is a mechanism to estimate the work needed to implement a change [18], points to software entities that would be changed [17], and aids in re-executing the right test cases to ensure that the new changes were correctly implemented [79]. Change impact analysis is essential for continually evolving systems to aid the implementation and evaluation of changes.

Briand et al. [26] and Wilkie and Kitchenham [111] have used coupling metrics to investigate object-oriented systems impact analysis. Wilkie and Kitchenham [111] found that high Coupling Between Object Classes (CBO) classes are more likely to be impacted by change ripple effects than low CBO classes. Briand et al. [26] examined utilizing coupling metrics to identify classes expected to be changed during impact analysis. Their experimentation showed that coupling metrics could be used to decrease the effort of impact analysis. Contrariwise, the study exposed an abundant amount of ripple effects that were not captured by highly structurally coupled classes. Poshyvanyk et al. [88] investigated using semantic coupling metrics during change impact analysis. They found that semantic coupling is a better predictor for classes impacted by changes compared to structural coupling. Since their metric (CCBCm) outperformed other coupling metrics in

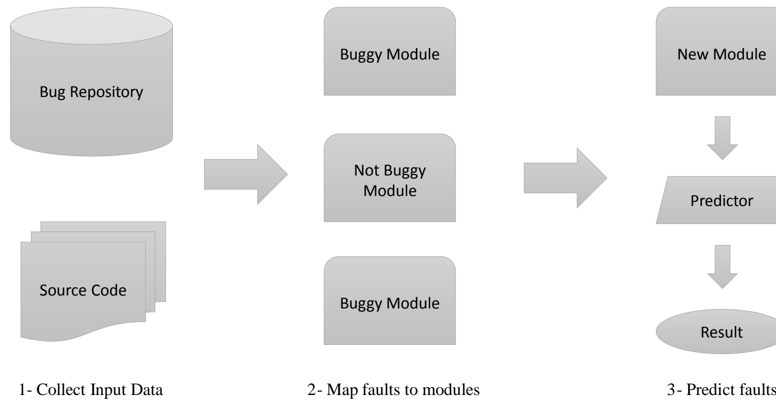

Figure 8. Prediction Using Historical Faults.

impact analysis, this dissertation follows their approach by comparing the new metric with their metric.

One possible area where an effort is beneficial to lower the maintenance costs is identifying the source code parts that are most likely to encompass faults, and therefore require changes. Software projects with available code and fault histories allow mapping their faults back to individual entities. The metrics for each entity involved in the mapping process then are computed. Relationships between metrics and faults are analyzed which results in a prediction model that can then be used to estimate the fault probability of new entities.

The general procedure behind fault prediction model development is depicted in Figure 8. The first step involves collecting the fault history of modules and calculating the software metrics from source code. The second step is mapping faults to modules by specifying which of the modules has a history of containing faults. The third step is building the predictive model to predict faulty modules in the next release.

Despite the fact that many fault prediction models have been suggested, still there is a need to have dependable tools that can be generally applied to real systems. For projects with fault histories, many research studies show that statistical models can deliver realistic approximations for predicting faulty entities using software metrics. There are abundant

research projects that built fault-proneness prediction models for object-oriented systems [12, 25, 38, 114, 49].

Basili et al. [28] began to consider using the CK metrics as early predictors for fault-prone classes [12]. They showed that CK metrics were statistically independent and can predict class fault-proneness. Tang et al. [100] used logistic regression to evaluate the performance of several product metric in predicting fault-proneness. Their results showed that WMC and RFC were significant indicators of fault-prone classes, but CK metrics alone were not satisfactory predictors of quality.

Briand et al. [25] conducted a case study on quality factors of object-oriented design. They found that a number of metrics from the CK metrics suite [28] were statistically associated with the fault-proneness of classes. El Emam et al. [38] used object-oriented metrics to identify fault-prone classes using prediction models. To construct a prediction model, they collected data from a commercial Java application. They stated that export coupling and inheritance depth were the best metrics to identify faulty classes.

Yu et al. [114] used a client side of a big network management Java-based system to examine the association between eight metrics and fault-proneness. They came to the conclusion that except DIT all CK metrics [28] were sound predictors of fault-prone classes. Gyimothy et al. [49] performed an empirical study that investigated eight object-oriented metrics for fault prediction of Mozilla. They found strong statistically significant correlations between most of the CK metrics [28] and fault-proneness.

Pai and Dugan [80] utilized several regression techniques. They have found that WMC, CBO, RFC, and LOC metrics were very effective in predicting the fault-proneness of classes.

Maintainability has come to be one of the foremost vital characteristics of software quality. Software maintenance is the most expensive and resource-intensive task of software development. Numerous research works showed that maintainability is not usually

addressed in the requirements and design stages [27]. Several problematic aspects have been shown as detrimental for software maintainability, such as lack of early thought of appropriate modularity selections. Several research studies established that better internal-quality classes (i.e., higher cohesion and lower coupling values) have better maintainability [31, 32, 65, 66].

Classes that undergo extensive changes are less maintainable than those with fewer modified LOC. Maintainability is an important external software attribute [42]. Low maintainability classes need to be documented very well to improve their understandability and cautiously tested to reduce their bugginess. Additionally, before releasing the system, low maintainability classes can be refactored to improve their maintainability. Refactoring these classes early is preferable since software developers have more knowledge about the software than maintenance developers have. Therefore, identifying classes with low maintainability is of great interest to the software engineering community.

Another substantial obstacle to maintenance is the interconnection between software entities [102]. Actually, there is an increasing empirical indication that the crucial characteristics of software maintainability are usually in reverse proportional to the presence of strongly coupled software entities [31, 32, 65, 66].

The most widely used maintainability dataset is the data collected by Li and Henry [65]. Li and Henry used two medium-sized Ada systems to study quality metrics that predict maintainability. Their results established a strong correlation between the quality metrics and class maintainability. Zhou and Leung [116] utilized Multivariate Adaptive Regression Splines (MARS) to predict the maintainability of object-oriented systems. They compared MARS with multivariate linear regression, artificial neural networks, regression trees, and support vector machines. Conversely, their technique does not outperform the compared techniques. Li-jin et al. [66] used Projection Pursuit Regression (nonparametric multivariate regression technique). Elish and Elish [39] used Tree Nets and showed that

they also provide competitive results when compared with other models utilizing the data collected by Li and Henry [65].

Dagpinar and Jahnke [31] predicted the maintainability of object-oriented classes using object-oriented metrics. They concluded that import coupling and size metrics are important maintainability predictors, whereas cohesion, inheritance, and export coupling metrics are not. Al Dallal [32] empirically studied the association between OO measures and the maintainability of classes. He used several statistical techniques to build models based on the selected metrics to predict class maintainability. The results demonstrated that classes with better internal qualities have better maintainability than those of worse internal qualities.

This dissertation bases maintainability on the same basis proposed by Dagpinar and Jahnke [31] and Li and Henry [65]. In addition, the results of the existing empirical studies [52] concerning the association found between maintenance cost and effort and the maintainability indicators are considered.

2.2. Related Work

Coupling measurement is an interesting and plentiful body of research work, which contains several coupling measurement approaches: structural coupling metrics [19, 28, 64], semantic coupling metrics [87, 105, 47], dynamic coupling metrics [8], evolutionary and logical coupling [44], information entropy coupling approach [6], coupling metrics for specific types of software applications such as procedural systems [75], knowledge-based systems [62], ontology-based systems [78] and aspect-oriented systems [115]. Since the new metric captures both structural and semantic relationships of software entities, a literature review is presented for both coupling metrics types.

2.2.1. Structural Coupling Metrics

Structural coupling metrics have received substantial consideration in the literature. Chidamber and Kemerer [28] introduced a metric suite to capture both coupling and co-

hesion in OO systems. They introduced the following coupling measures: Depth of Inheritance Tree (DIT), Coupling between objects (CBO) and response for a class (RFC). The DIT metric offers for each class a measure of the levels of inheritance from the object hierarchy top. For CBO, two classes are coupled when methods in one class use methods or fields defined by the other class. RFC is defined as the number of methods that can be potentially executed in response to a message received by an object of that class. In other words, RFC counts the number of methods invoked by a class.

Afferent coupling (Ca) and efferent coupling (Ce), that use the term category, which is a set of classes that achieve some common goal (i.e., package), were identified by Martin [72]. Ca is the number of classes outside the category that depend upon classes within the category, while Ce is the number of classes inside the category that depend upon classes outside the categories.

Bansiya et al. [11] proposed Number of Public Methods (NPM) which counts all publicly defined methods in a class. Tang et al. [100] proposed Inheritance Coupling (IC) and Coupling Between Methods (CBM). IC counts the number of parent classes to which a given class is coupled. If one of the classes inherited methods functionally dependent on the redefined methods in another, it is considered coupled to its parent class. For the class to be considered coupled one of several criteria should be satisfied: one of its inherited methods uses an attribute that is defined in a new/redefined method; one of its inherited methods calls a redefined method; or one of its inherited methods is called by a redefined method and uses a parameter that is defined in the redefined method. CBM represents the number of inherited redefined/new methods, which are coupled. The coupling exists when at least one conditions is held.

2.2.2. Semantic Coupling Metrics

For the semantic relationships of software entities, Poshyvanyk and Marcus [87] presented a new set of conceptual coupling metrics. They outlined a coupling metric for

classes which utilizes source code textual information, $CoCC$ (Conceptual Coupling of Classes). Using open source C++ software systems, they showed that the semantic coupling captured a new dimension of coupling not captured by structural metrics. Ujhazi et al. [105] added an empirically derived threshold to distinguish between strong and weak conceptual similarities and came up with a new semantic metric namely CCBO (Conceptual Coupling between Object Classes). Another semantic class coupling metric was proposed by Gethers and Poshyvanyk [47], the Relational Topic-based Coupling (RTC) metric. Their metric uses a variant of Latent Dirichlet Allocation (LDA) called Relational Topic Models (RTM). LDA is a generative model that allows sets of words to be explained by unobserved groups that explain why some parts of the data are similar. RTM is an extension to LDA in which the links are explicitly modeled between documents in the corpus. These embedded links explains both the words of the documents and how they are connected. RTC utilizes these links to describe the coupling between two classes in the corpus. The authors demonstrated that the new metric is statistically different from existing metrics. Revelle et al. [91] introduced Textual Feature Coupling (TFC) which quantifies the coupling between two features relying on semantic information in source code using Latent Semantic Indexing (LSI).

CHAPTER 3. COMBINING STRUCTURAL AND SEMANTIC COUPLING

In this chapter, a new coupling metric is introduced that measures the structural and semantic relationships among methods and classes. It is noteworthy that previous studies [87, 88] found no correlation between structural and semantic coupling metrics, which indicates that they measure different coupling aspects. Orthogonality of coupling metrics may present the opportunity for improved coupling measurement through combining different measures. The combined metric allows the analysis of the relationship from a structural perspective and a semantic perspective. The conjecture is that better coupling comprehension is achieved through combining structural and semantic coupling relationships.

Both structural (i.e., structured information) and semantic (i.e., unstructured information) relationships can be extracted from the source code. Structured information refers to structural relations between software entities. Examples of structured data include source code parse trees, control flow, call graphs, and inheritance graphs. Unstructured information denotes source code comments and identifier names that embody domain knowledge. The unstructured portion of source code has been shown to help determine the high-level functionality of the source code [63].

The remainder of this section delivers details on how structural and semantic information are extracted from software source code. The chapter is organized as follows: section 3.1 presents the structural coupling relationships used in the new metric; section 3.2 presents the semantic coupling relationships used in the new metric; section 3.3 presents the new combined coupling metric; section 3.4 presents a toy example of measuring the new coupling metric; and section 3.5 presents the details of the tool implementation to calculate this new metric.

3.1. Structural Coupling Relationships

Structural coupling relationships comprise the set of all relationships among compilation units present in the source code. These include calls from method to another, uses of common variable, etc. Structural dependencies occur whenever a compilation unit depends on another during compilation. Strictly speaking, A depends on B when the source code of A makes an explicit reference to B. In such a case, in order to compile A to an executable form, it is necessary to have access to B. For more detailed information about software dependencies please refer to Section 2.1.3.

An important feature of structural coupling is the direction of the dependency. The structural coupling between a software entity, such as a class or a method, and its surrounding environment (i.e., the other entities in the system) may be inbound or outbound [22]. The counting of inbound and outbound dependencies of a method represents the importance of that method in the system (i.e., how much this method is coupled to the system). Method invocations are hypothesized to be the most important type of coupling connections [22].

This dissertation concentrates on methods as the main unit for coupling. Methods implement functionality in the source code. Methods that use other methods in the same class in which the method is defined versus using methods defined in other classes require different developer actions. This distinction is essential due to the increased amount of effort required by a developer to use methods that are defined in a different class. In such a situation, the developer must get hold of the public interface of the class that contains the method, choose a message from the interface, and decide what available variables are suitable arguments for the message [10]. Consequently, it is more difficult to use a method defined outside the current class.

Since the method is the basis of the measurement, the following are considered structural method coupling: inbound method dependencies (methods in other classes that depend on this method), outbound method dependencies (methods in other classes that

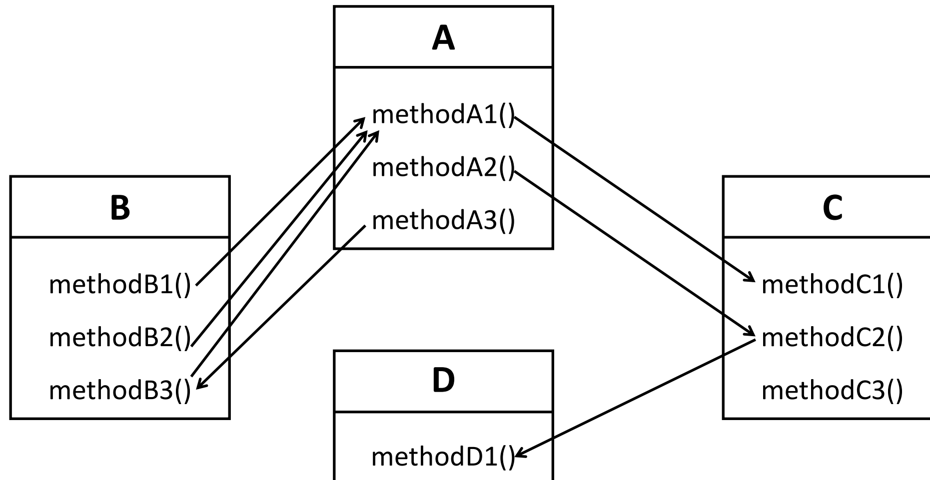

Figure 9. An Illustration of Structural Coupling.

this method depends on), and if these two method directly depend on each other. In the case of method-pair coupling, the fact that these method-pair are directly coupled increases their dependencies. The number of dependencies of a method represents how much this method is coupled to other methods in the system. Two methods are directly coupled (method-method interaction coupling) in two cases: they communicate via sharing of data, or one method invokes the other. Figure 9 shows an illustration of structural coupling (For example, methodA1() has four dependencies, methodB1() has one dependency, methodC1() has one dependency, and there is a direct dependency between the method-pair (methodA2(),methodC2()).

3.2. Semantic Coupling Relationships

Analyzing the unstructured information in the source code (that is, the comments and identifiers) is based on the idea that the unstructured information reveals, to some extent, the concepts of the problem domain of the software. This information adds a new layer of source code semantic information and captures the domain semantics of the software [68, 71, 70, 87].

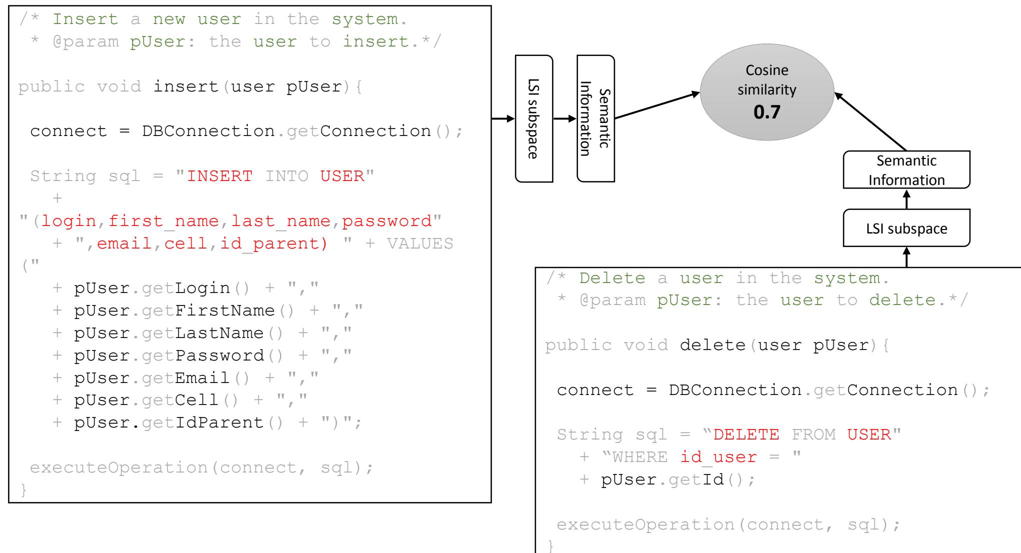

Figure 10. An Illustration of Semantic Coupling.

Developers use identifiers and comments to embody elements of the solution domain of the software. This type of information is used usually in supporting maintenance and evolution [16], program comprehension [67], and coupling [87, 105]. The semantic relations between software entities may exist, as they possibly will contribute together to the implementation of a domain concept. For example, if two methods use similar identifier names and contain similar comments, they implement similar domain concept. The semantic coupling between software entities can be captured by measuring the semantic similarity between their textual contents (after preprocessing) [87, 105]. The preprocessing steps are disregard language keywords, split identifiers, and remove stop words. The cosine similarity measures the semantic information found in the identifiers and comments of the software entity. The similarity of two documents is the cosine value between their corresponding vectors that increases as more terms are shared [70, 87, 105].

To infer the semantic relations, Latent Semantic Indexing (LSI) is used. LSI detects semantic relationships between words and concepts in textual information. LSI has been widely used for several software engineering applications such as feature location [30, 92], traceability link recovery [56], software measurement [70, 88], and detecting code clones

[99]. This semantic similarity measures the semantic information shared between methods of different classes. Two methods are semantically related if their (domain) semantics are similar (i.e., they use similar vocabulary). If two methods have a high semantic similarity and use similar terminology, they might implement similar concepts and consequently be coupled [87, 105]. For more detailed information about LSI and cosine similarity, please refer to Section 2.1.4. Figure 10 shows an illustration of semantic coupling.

3.3. Combined Coupling Metric

A representation for a software system is defined. S represents an object-oriented system. S has a set of classes $C = \{c_1, c_2, \cdots, c_n\}$. The number of classes in the system is $n = |C|$. A class has a set of methods. For each class $c \in C, M(c) = \{m_1, m_2, \cdots, m_t\}$ are the methods in c, where $t = |M(c)|$ is the number of methods in a class c. The set of all methods in the system S is denoted by $M(S)$.

$$MPC(m_i, m_j | c_k, c_l) = \frac{(MD(m_i, m_j))(1 + DEP(m_i, m_j))(1 + COS(vm_i, vm_j))}{AVG} \quad (1)$$

$$DEP(m_i, m_j) = \begin{cases} 1 & \text{if } m_i \text{ depends on } m_j \text{ or } m_j \text{ depends on } m_i \\ 0 & \text{otherwise} \end{cases} \quad (2)$$

$$AVG = \frac{\sum_{t=1}^{|M(c_k)|} \sum_{s=1}^{|M(c_l)|} (MD(m_{kt}, m_{ls}))(1 + DEP(m_{kt}, m_{ls}))(1 + COS(vm_{kt}, vm_{ls}))}{|M(c_k)| * |M(c_l)|} \quad (3)$$

m_i is a method $\in$ class k (c_k), and m_j is a method $\in$ class l (c_l). m_i, m_j is the method pair. MD represents the number of dependencies of a method. A dependency

is a relationship between two methods where one uses the services provided by the other (e.g., method m_i calls method m_j). vmi is the LSI representation of the textual content of method m_i as explained in Chapter 2.

Basically, Method Pair Coupling (MPC) consists of four different components. The four components are justified as follows:

- The first component is $(MD(m_i, m_j))$. MD represents the number of dependencies of the two studied methods. This components emphasizes the importance of these methods (how much these methods are coupled to the rest of the system).

- The second component is $(1 + DEP(m_i, m_j))$. The DEP value is either 0 or 1. If there is a direct dependency between the two methods, then DEP is 1, 0 otherwise. The rational of this component is to strengthen the coupling value between the two methods if they have a direct dependency between them.

- The third component is $(1 + COS(vm_i, vm_j))$. $COS(vm_i, vm_j)$ represents the cosine similarity of the two methods represented in LSI. Usually, the cosine value is between -1 and 1. However, In this dissertation, any negative cosine value is replaced by zero. Since the cosine value now ranges between 0 and 1, adding one to that value is for the purpose of increasing the measure in case of having strong semantic similarity (i.e., multiplying by a value less than 1 decreases the value). The cosine similarity here serves as an incorporation of the semantic relations.

- To normalize the measure, AVG is used to normalize the measure at the hybrid method pair coupling (average of all method pairs in the two subjected classes). Dividing by the average is an example of rescaling the data, which is usually done to data that do not depend on measurement units. This rescaling operation is called normalizing the data to the mean, which makes it possible to be efficiently compared against other values.

Coupling is a quantity of dependencies between a software entity and the rest of the system [22]. Coupling is a ratio scale because any coupling reduction is given in ratios and there is a natural zero level of coupling when modules are not related to each other [1]. Zuse in his framework for software measurement [117], mentions that if we have two measures on the ratio scale and they are logically independent (do not correlate), then the hybrid measure by multiplication can be used as a ratio scale measure. Since MPC is a ratio scale measure, multiplication and division are valid operations [1].

MPC combines several coupling relationships between methods in different classes, where each measure reflects a different type of relationship between methods. The contributions of each measurement components are combined into a unique value reflecting the overall structural and semantic similarity of both methods. Then this combination is normalized to the mean, making it possible to be efficiently compared against other values.

The product of structural and semantic represents all possible combinations of both of them. As these combinations increase, so too, should the coupling. A product is the result of multiplying different terms. For instance, 15 is the product of 3 and 5. The numbers 3 and 5 are surely independent since the number 3 does not depend on the number 5. However, the result of the multiplication depends on both numbers. In this kind of combination, the values of the hybrid measures double if the value of the single measure is doubled. This denotes that they are logically independent, which means the measurement value of the first measure should not be changed by changing the second measure value [117]. The concept of a product describes how to combine two objects of some kind to create an object, possibly of a different kind.

MPC calculates the number of dependencies of the method pair in different classes (MD). The value will increase if there is a direct dependency between the method pair (DEP). The value also increases if there is a strong cosine similarity of the methods' representations ($COS(vm_i, vm_j)$) since large cosine values indicate more similar methods.

After calculating the method-pair coupling, the next step is to calculate the method-to-class, then class-to-class, and finally class-level coupling [87, 105, 91]. Hybrid Coupling between method and a class (HCMC) is the average of the measure between method m_i and all the methods from class c_j.

$$HCMC(m_i, c_j) = \frac{\sum_{q=1}^{t} MPC(m_i, m_{jq})}{t} \tag{4}$$

where t is the number of methods in c_j.

The coupling between two classes is defined $c_i \in$ S and $c_j \in$ S as:

$$HCCC(c_i, c_j) = \frac{\sum_{l=1}^{t} HCMC(m_{il}, c_j)}{t} \tag{5}$$

where t is the number of methods in c_i. Hybrid Coupling between two classes (HCCC) is the average coupling measure between all unordered pairs of methods from class c_i and class c_j. This equation assures symmetrical coupling between the two classes as $HCCC(c_i, c_j) = HCCC(c_j, c_i)$.

To measure the hybrid coupling at the class level, a coupling metric is defined that estimates the coupling of a class in an object-oriented software system (SSCM). The class coupling measures the degree to which the methods of a class are coupled to the methods of other classes in the system.

$$SSCM(c_i) = \frac{\sum_{k=1}^{n} HCCC(c_i, c_k)}{n-1} \tag{6}$$

where n is the number of classes.

3.4. A Toy Example of Measuring the New Coupling Metric

To demonstrate how SSCM is calculated, let us consider three classes: $C_1 = \{m_1, m_2\}$, $C_2 = \{m_3, m_4, m_5\}$ and $C_3 = \{m_6, m_7, m_8\}$. In order to compute the new coupling metric, three inputs should be provided: (1) structural method dependencies, (2) direct dependencies between methods, and (3) the cosine similarities between methods.

The method dependencies of these three classes are shown in Table 3. These dependencies represent the number of method dependencies from or to this method. Table 3 shows the direct dependencies between methods. Two methods are directly coupled in two cases: they communicate via sharing of data, or one method invokes the other. 1 means there is a direct dependency and 0 means there is not. The cosine similarities between these methods are outlined in Table 5.

Table 3. Toy Example: Method Dependencies

Method	# dependencies
m_1	2
m_2	3
m_3	2
m_4	2
m_5	4
m_6	2
m_7	1
m_8	1

Table 4. Toy Example: Direct Dependencies Between Methods

	m_1	m_2	m_3	m_4	m_5	m_6	m_7	m_8
m_1		0	0	0	0	0	0	0
m_2			1	1	0	0	0	0
m_3				0	0	0	0	1
m_4					0	0	0	0
m_5						0	1	0
m_6							0	0
m_7								0

Table 5. Toy Example: The Cosine Similarities Between Methods

	m_1	m_2	m_3	m_4	m_5	m_6	m_7	m_8
m_1	1	0.5	0.4	0.25	0.11	0	0.41	0.65
m_2	0.5	1	0.58	0.24	0.27	0.43	0.39	0.44
m_3	0.4	0.58	1	0.45	0.39	0.56	0.66	0.21
m_4	0.25	0.24	0.45	1	0.34	0.47	0.23	0.18
m_5	0.11	0.27	0.39	0.34	1	0.05	0.03	0.5
m_6	0	0.43	0.56	0.47	0.05	1	0.23	0.43
m_7	0.41	0.39	0.66	0.23	0.03	0.23	1	0.54
m_8	0.65	0.44	0.21	0.18	0.5	0.43	0.54	1

Since all the inputs are available now, computing the metric is possible. First, method-pair coupling is computed (MPS) between different classes. Table 6 shows the MPC numerator values for method-pairs in classes $C_1 and C_2$. To calculate the AVG value for normalization, AVG equals the average of all method pairs between these two classes: (5.6+5+6.66+15.8+12.4+8.89) / 6 = 9.05, AVG = 9.05.

Table 6. Toy Example: MPC numerator values for method-pairs in C_1 and C_2

	m_1	m_2	m_3	m_4	m_5
m_1		7.5	5.6	5	6.66
m_2			15.8	12.4	8.89
m_3				5.8	8.34
m_3					8.04

To illustrate more how MPC is calculated, three examples are shown. To measure the hybrid coupling between m_1 and m_3: MPC(m_1, m_3) = (4)*(1+0)*(1+0.4) / 9.05. MPC(m_1, m_3) = 5.6 / 9.05. MPC(m_1, m_3) = 0.62. The hybrid coupling between m_1 and m_4: MPC(m_1, m_4) = (4)*(1+0)*(1+0.25) / 9.05. MPC(m_1, m_4) = 5 / 9.05. MPC(m_1, m_4) = 0.55. The hybrid coupling between m_1 and m_5: MPC(m_1, m_5) = (6)*(1+0)*(1+0.11) / 9.05. MPC(m_1, m_5) = 6.66 / 9.05. MPC(m_1, m_5) = 0.74. These examples shows that how having different semantic similarities affects the method-pair coupling value (0.4 cosine similarity is more than 0.25 which resulted in MPC(m_1, m_3) = 0.62 and MPC(m_1, m_4) = 0.55.

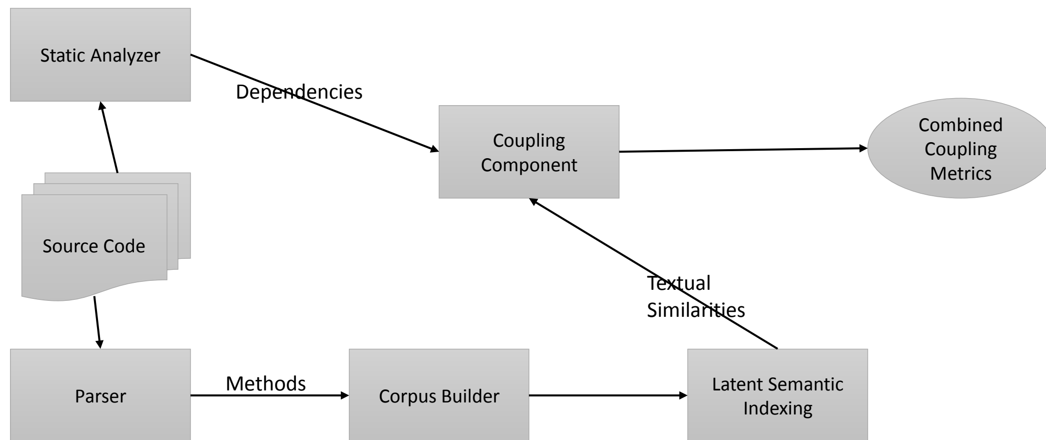

Figure 11. Architecture of the Combined Coupling Component.

An example of measuring a hybrid coupling with a direct dependency, the hybrid coupling between m_2 and m_3: MPC(m_2, m_3) = (5)*(1+1)*(1+0.58) / 9.05. MPC(m_2, m_3) = 15.8 / 9.05. MPC(m_2, m_3) = 1.75.

Next, hybrid coupling between method and a class (HCMC) is computed. For example, measuring the hybrid coupling between method m_1 and class C_2: HCMC(m_1, C_2) = MPC(m_1, m_3) + MPC(m_1, m_4) + MPC(m_1, m_5) / number of methods(C_2). HCMC(m_1, C_2) = 0.62 + 0.55 + 0.74 / 3. HCMC(m_1, C_2) = 0.64.

After that, the hybrid coupling between two classes (C_1 and C_2) is computed: HCCC(C_1, C_2) = HCMC(m_1, C_2) + HCMC(m_2, C_2) / number of methods(C_1). HCCC(C_1, C_2) = 1.36.

Finally, the hybrid coupling of a class (SSCM) is computed. The hybrid coupling of classes (C_1 and C_2) is computed as: SSCM(C_1) = 0.64 and SSCM(C_2) = 1.27.

3.5. Tool Implementation

A tool has been developed that calculates the new metrics and the CoCC metric automatically (Figure 11). The tool extracts the structural and semantic connections between methods. The tool was developed using the R language version 2.15.1[1].

[1]http://www.r-project.org/

Starting from the source code, the static analyzer (DependencyFinder [101]) extracts the compile-time dependences. These compile-time dependencies are stored in an XML file. A Java-based parser was developed to extract the textual contents of the methods in a system. Each method in the systems in represented as a document.

The corpus builder component builds the corpus and preprocesses the text. The preprocessing steps include disregard language keywords, split identifiers, and remove stop words. The LSI component creates the LSI subspace of these methods using the 'lsa' package[2]. It also calculates the cosine similarity between each method pair in the system.

The coupling component reads the structural relationships from the XML file produced by the static analyzer and the semantic relationships from the LSI component. It also calculates the the new metrics based on the aforementioned equations and the CoCC metric.

[2]http://cran.r-project.org/web/packages/lsa/index.html

CHAPTER 4. ASSESSMENT OF THE NEW METRIC

This chapter provides statistical and analytical evaluation of the new coupling metric to study its validity [59]. Statistical validation tests whether what is being measured exhibits expected statistical relationships with other measures. Analytical validation investigates if the new metric satisfies necessary properties of the measured attribute. Consistent with this general validation approach, SSCM is then examined from both statistical and analytical standpoints. An empirical evaluation of the new metric is detailed in Chapter 5.

The chapter is organized as follows: section 4.1 presents the statistical evaluation of the new metric; and section 4.2 presents the analytical evaluation of the new metric.

4.1. Statistical Evaluation

This section investigates whether the new proposed coupling metric measures something different than the other coupling metrics.

4.1.1. Coupling Metrics

To investigate the statistical relationships between the new metric and existing coupling metrics, several existing coupling metrics are selected for evaluation: DIT, CBO, RFC [28], Ca, Ce [72], NPM [11], IC, CBM [100], and CoCC [87] (Table 7). The metrics come from several metrics suites. These coupling metrics are widely used and cover several aspects of coupling: inheritance, interaction, component, and semantic. A detailed description of the selected coupling metrics is given as follows:

Depth of inheritance tree of a class (DIT): The metric measures the length of the longest path of inheritance structure finishing at the current class. DIT measures how many antecedent classes can possibly affect this class. The deeper the inheritance tree for a class, the more difficult it might be to predict its behavior.

Coupling between objects (CBO): The metric measures the number of other classes that are coupled to the current class. The coupling can happen through field accesses, inheritance, arguments, method calls, return types, and exceptions.

Table 7. Coupling Metrics

Abbri.	Metric	Reference
DIT	Depth of Inheritance Tree	[28]
CBO	Coupling between object classes	[28]
RFC	Response for a class	[28]
Ca	Afferent couplings	[72]
Ce	Efferent couplings	[72]
NPM	Number of Public Methods	[11]
IC	Inheritance Coupling	[100]
CBM	Coupling Between Methods	[100]
CoCC	Conceptual Coupling of Classes	[87]

Response for a class (RFC): The metric measures the number of methods that can possibly be executed in response to a message received by an object of that class. Because it includes methods called from outside the class, it is also a measure of the potential communication between the class and other classes.

Afferent Couplings (Ca): The metric measures the number of classes from other packages depending on classes in the analyzed package. This is related to the packages responsibility. Ca is the number of other packages depending on one package. A high number indicates bad design, or that the package is used for crosscutting concerns.

Efferent Couplings (Ce): The metric measures the number of packages the classes of the analyzed package depend upon. This relates to the packages independence.

Number of Public Methods (NPM): The metric counts all the publicly defined methods in a class. A high NPM value indicates two different bad smells in the design of a software. First, a signal for a complex class that has too many responsibilities. Second, a signal for a highly coupled class with other parts of the software, because every public method may expose the internally used classes.

Inheritance Coupling (IC): The metric measures the number of parent classes to which a given class is coupled. A class is coupled to its parent class if one of its inherited methods is functionally dependent on the new or redefined methods. A class is coupled to

its parent class in one of the following ways: if one its inherited methods uses a variable (or data member) that is defined in a new/redefined method; if one of its inherited methods calls a redefined method and uses the return value of the redefined method; if one of its inherited methods is called by a redefined method and uses a parameter that is defined in the redefined method; or if one of its inherited methods uses a variable and the value of that variable depends on the value of another variable, which is defined in a new/redefined method. The underlying assumption of the IC metric is that when a data member, which is used by an inherited method, changed by a new or redefined method, it is might introduce new faults into the inherited method.

Coupling Between Methods (CBM): The metric measures the total number of new/redefined methods to which all the inherited methods are coupled. An inherited method is coupled to a new/redefined method if it is functionally dependent on a new/redefined method in the class. Therefore, the number of new/redefined methods to which an inherited method is coupled can be measured. The assumption behind the CBM metric is that IC is too limited since it only measures the number of parent classes to which a given class is coupled, without the CBM, additional function dependency complexity at the methods level is not considered.

Conceptual Coupling of Classes (CoCC): It represents the semantic information obtained from the source code, embodied in comments and identifiers. It measures the textual similarity between classes through measuring the similarity between the textual contents of their methods.

4.1.2. Subject Software Systems

The subject systems are limited to open source software systems developed using Java. To select the systems, four selection criteria have been used. First, the selected systems had to be well-known systems that are used very widely. Second, the systems had to be sizable, so the systems can be realistic and have multiple developers. Third, the

systems had to be actively maintained. Finally, the data of these systems had to be publicly available. Furthermore, the faults of all these systems have been collected and are publicly available at the PROMISE repository[3]. Six various-sized systems have been chosen from different domains. Characteristics of the selected software systems are listed in Table 8.

Table 8. Selected Software Systems

System	Ver	Packages	Classes	Methods	KLOC	Source
POI	3.0	16	386	6302	129.33	http://poi.apache.org/
Synapse	1.2	29	219	1875	53.5	http://synapse.apache.org/
Lucene	2.4	13	283	4211	102.86	http://lucene.apache.org/
Ivy	2.0	28	351	2806	87.77	http://ant.apache.org/ivy/
Log4j	1.2	22	156	1679	83.19	http://logging.apache.org/log4j/
Xerces	1.4.4	19	588	5896	141.18	http://xerces.apache.org/

4.1.3. Measurement Results

The structural coupling metrics values were downloaded from the PROMISE repository whereas CoCC and the new metric were computed using our own tool. Table 9 shows descriptive statistics about the coupling metrics.

Table 9. Descriptive Statistics for the Coupling Metrics

Measures	Min	Max	Med	σ
DIT	1	7	2	1.07
CBO	0	214	7	14.11
RFC	1	390	22	34.01
Ca	0	212	2	11.92
Ce	0	133	4	7.74
NPM	0	142	6	11.01
IC	0	4	0	0.68
CBM	0	21	0	2.59
CoCC	0	0.94	0.28	0.08
SCM	0	9.79	0.97	0.45
SSCM	0	19.54	0.75	0.91

To understand the relationships between software metrics, their correlation coefficient (the strength of relationship among their counterparts) can be measured. Table

[3] https://code.google.com/p/promisedata/

Table 10. Pearson Rank Correlations Between Coupling Metrics

	DIT	CBO	RFC	Ca	Ce	NPM	IC	CBM	CoCC
DIT									
CBO	-0.16								
RFC	-0.11	0.51							
Ca	-0.14	0.86	0.18						
Ce	-0.14	0.56	0.70	0.10					
NPM	-0.03	0.34	0.69	0.22	0.31				
IC	0.47	-0.04	0.06	-0.03	-0.05	0.10			
CBM	0.40	-0.05	0.14	-0.05	0.00	0.21	0.73		
CoCC	-0.15	0.05	0.02	0.00	0.10	-0.09	-0.12	-0.13	
SSCM	-0.04	0.01	0.01	-0.01	0.03	0.02	-0.01	-0.02	-0.02

10 shows the Pearson rank correlations between coupling metrics. Correlations greater than 0.65 are highlighted since they indicate strong correlation. Table 10 shows that the new coupling metric (SSCM) does not correlate with any of the coupling metrics, which indicates that it measures something that is not measured by the other coupling metric. The Table also shows that the following coupling metrics are strongly correlated with each other:

- Ca strongly correlates with CBO. This strong correlation can be explained since both of them count inbound coupling.

- Ce strongly correlates with RFC. This strong correlation can be explained since both of them capture outbound coupling.

- NPM strongly correlates with RFC. This strong correlation can be explained since both of them capture the number of accessible methods.

- CBM strongly correlates with IC. This strong correlation can be explained since a class is coupled to its parent class (in case of IC) if one of its inherited methods is functionally dependent on the new or redefined methods, while CBM is the total number of new/redefined methods.

4.1.4. Cluster Analysis

Cluster analysis (CA) is a multivariate technique, with the aim of grouping objects into groups in such a way that the degree of association between two objects is maximal if they belong to the same group and minimal otherwise [103]. CA is one of the main techniques for statistical data analysis.

The coupling metric data of six open source projects is high-dimensional data with many data vectors. CA is used here to group similar coupling metrics. In particular, hierarchical clustering with p-values via multi-scale bootstrap re-sampling [96] is used to group similar coupling metrics. The coupling metric data of six open source projects is high-dimensional data with many data vectors. CA is used here to group similar coupling metrics. In particular, hierarchical clustering with p-values via multi-scale bootstrap re-sampling [96] is used to group similar coupling metrics. The distance used is correlation and the cluster method is average. This type of hierarchical clustering computes a bootstrap probability (BP) and an approximately unbiased (AU) p-value for each cluster. These p-values indicate how strong a clustering is supported by the data. The distance used in the clustering is correlation and the clustering method is average. The implementation of this clustering technique is available in the 'pvclust' package[4]. The goal here is to see whether the new coupling metric (SSCM) is grouped with any one of the other coupling metrics. The assumption is that the new coupling metric is a combination of structural and semantic relationships, which will lead to grouping it in a separate cluster.

Figure 12 shows the resulting clusters of the coupling data. The left side shows DIT, IC, and CBM in one cluster. These coupling metrics have inheritance relationships. The right side shows CBO, Ca, NPM, RFC, and Ce. These coupling metrics represent interaction coupling (inbound/outbound) The center side shows SSCM in one cluster, which indicates that it measures a different kind of coupling than other coupling metrics.

[4]http://cran.r-project.org/web/packages/pvclust/

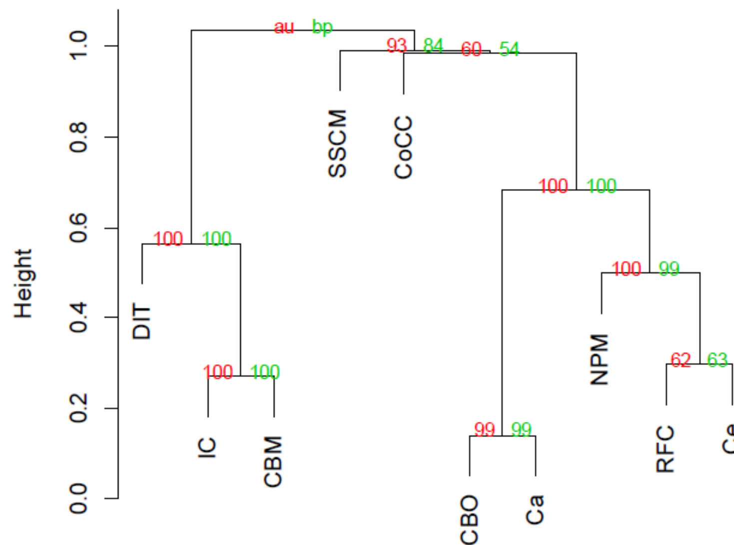

Figure 12. Cluster Dendrogram of the Coupling Data.

RFC and Ce were grouped together since both of them capture outbound coupling. Ca and CBO were grouped together since both of them count inbound coupling. IC and CBM were grouped together since a class is coupled to its parent class (in case of IC) if one of its inherited methods are functionally dependent on the new or redefined methods. SSCM was grouped as a separate group which indicates that SSCM is different (measure something different) than the other coupling metrics. The fact that CoCC is grouped by itself in a different cluster is consistent with previous studies [87, 105].

4.2. Analytical Evaluation

Many existing software metrics are criticized from two standpoints, theoretically and empirically. Several researchers pointed that most software metrics were developed with no or little theoretical basis [24, 40, 85]. Furthermore, even though some metrics are theoretically valid, they lack empirical evaluation [7, 21, 40].

There are several frameworks for the analytical evaluation of metrics. Some of them are mainly subjective, while others rely on either axiomatic or measurement theory foundations. The widely known frameworks are: property-based [24], distance-based [85],

and Schneidewind's metrics validation methodology [95]. According to Kitchenham et al. [60] property-based approaches can be used as a mechanism to reject proposed metrics, but they are not satisfactory to prove the validity of the metric. Unlike mathematical axioms, these axioms serve only as a filter to reject things which are self-evidently not coupling measures.

Briand et al. property-based framework [24] was preferred since it is generic, comprehensive and based on measurement theory. Their framework is important because it explains the fundamental criteria that must be fulfilled by the various metrics. The framework has been utilized before to validate several metrics [84, 93, 87, 29]. It precisely specifies the mathematical properties that characterize coupling. The properties are (1) Nonnegativity; (2) Null Value; (3) Monotonicity; (4) Merging of Modules; and (5) Disjoint Module Additivity [24]. The objective of this evaluation is to determine whether the new coupling metric (SSCM) complies with the coupling properties proposed by Briand et. al [24].

The coupling properties are defined for a modular software system:

- S is a system with a tuple $< E, R >$, where E has elements and R $\subseteq$ E $\times$ E are relationships between the elements of E.
- MS is a modular system with a triple $< E, R, M >$, where $< E, R >$ is a system and M is a collection of modules that partitions E.
- A system m = $< Em, Rm >$ is a module of MS if and only if $E_m \subseteq$ E and $R_m \subseteq$ R.
- The modules of M are disjoint, i.e., for modules m_1 = $< Em_1, Rm_1 >$ and m_2 = $< Em_2, Rm_2 >$ of MS it must hold that $Em_1 \cap Em_2 = \varnothing$ (and hence $Rm_1 \cap Rm_2 = \varnothing$).
- The inter-modular relationships that a module m = $< E_m, R_m >$ of MS is involved in are captured in the sets InputR(m) and OutputR(m):

InputR(m) = $< e_1, e_2 > \in R \mid e_2 \in E_m$ and $e_1 \in E - E_m$;

OutputR(m) = $< e_1, e_2 > \in R \mid e_1 \in E_m$ and $e_2 \in E - E_m$.

OuterR(m) denotes either InputR(m) or OutputR(m) and is used to reflect the perspective (inbound/outbound) of the coupling measure.

- All intra-modular relationships in MS is IR = U R_m.

- All inter-modular relationships in MS is R IR.

The coupling of a [module m = $< E_m, R_m >$ of a modular system MS|modular system MS] is a function [Coupling(m)|Coupling(MS)] characterized by the following properties:

- **Property 1: Nonnegativity**:

  Given a module m = $< E_m, R_m >$ of a modular system MS = $< E, R, M >$

  [Coupling(m) ≥ 0|Coupling(MS) ≥ 0]

  This property states that the coupling of a software entity should not be a negative value. The equation of the new metric ensures that there is no negative value for any of the systems classes. The new metrics can equal zero or some positive value. It will never be negative under any conditions.

- **Property 2: Null Value**:

  Given a module m = $< E_m, R_m >$ of a modular system MS = $< E, R, M >$. When OuterR(m) is empty, the coupling of m is null. When R IR is empty (i.e., no inter-modular relationships), the coupling of MS is null.

  [OuterR(m) = $\varnothing \Rightarrow$ Coupling(m) = 0 | R - IR = $\varnothing \Rightarrow$ Coupling(MS) = 0]

  This property states that if an entity has no elements, then its coupling value should be zero. In other words, the coupling of a class with no external relationship is null.

If one of the classes has no elements or there is no relationship between two classes, then the coupling is zero between these classes. The equation of the new metric ensures that there is no null value for any of the systems classes. In case of having no relationships between two classes, the coupling value will be zero.

- **Property 3: Monotonicity**:

  Let $MS' =< E, R', M' >$ and $MS'' =< E, R'', M'' >$ be two modular systems (with the same set of elements E) such that there exist two modules $m' \in M'$ and $m'' \in M''$ such that $R' - OuterR(m') = R'' - OuterR(m''), andOuterR(m') \subseteq OuterR(m'')$. Then

  $[Coupling(m') \leq Coupling(m'') \mid Coupling(MS') \leq Coupling(MS'')]$

  This property states that adding inter-class relationships does not decrease coupling. If an extra relationship is added between a pair of classes whose inter-class coupling is to be measured, then the coupling between the classes must not decrease. The SSCM coupling cannot be decreased by adding more relationships between two classes. Since SSCM captures structural and semantic relationships, any added relationships between these two classes will be reflected in the measurement value.

- **Property 4: Merging of classes**:

  Let $MS' =< E, R, M' >$ and $MS'' =< E, R, M'' >$ be two modular systems (with the same underlying system $< E, R >$) such that $M'' = M' - m'_1, m'_2 \cup m''$, with $m'_1 \in M', m'_2 \in M', m'' \notin M'$ and $E_{m''} = E_{m'1} \cup E_{m'2}$ and $R_{m''} = R_{m'1} \cup R_{m'2} \cup \{ < e_1, e_2 > \in R \mid (e_1 \in E_{m'1}$ and $e_2 \in E_{m'2})$ or $(e_1 \in E_{m'2}$ and $e_2 \in E_{m'1})\}$. Then, $[Coupling(m'_1) + Coupling(m'_2) \geq Coupling(m'') \mid Coupling(MS) \geq Coupling(MS'')]$

  This property states that coupling of an entity should be no less than the sum of the couplings of any two of its modules with no relationships in common. Merging two

classes together will result in a reduced amount of coupling (or remain the same in the worst case), but it will not increase. When connected classes are merged, the cosine similarity remains the same because the repositioning of the methods inside other classes will not affect the semantic relatedness of these methods with methods of other classes. In addition, the dependencies of these methods will be the same.

- **Property 5: Disjoint class additivity**:

  Let $MS' =< E, R, M' > andMS'' =< E, R, M'' >$ be two modular systems (with the same underlying system $< E, R >$) such that $M'' = M' - m'_1, m'_2 \cup m''$, with $m'_1 \in M', m'_2 \in M', m'' \notin M'$ and $m'' = m'_1 \cup m'_2$. Then, $[Coupling(m'_1) + Coupling(m'_2) = Coupling(m'') \mid Coupling(MS) = Coupling(MS'')]$

  This property states that the coupling value obtained by merging two unrelated classes is equal to the sum of the couplings of the two original classes. When two or more classes having no relationships between them are merged, coupling cannot increase since the structural dependencies and semantic relatedness of methods of the unconnected classes will be the same no matter where they are placed.

CHAPTER 5. EMPIRICALLY EVALUATING THE NEW METRIC

This chapter empirically investigates the usefulness of the new coupling metric in different applications. The purpose of the investigation is to determine the practicality of the new coupling metric specifically regarding software maintenance. Three investigations have been executed of the proposed metric, each targeting a different aspect of software maintenance: change impact analysis, predicting faulty classes, and predicting maintainable classes. In the view of the investigation results, a robust conclusion can be derived about the practicality of the new coupling metric.

An important part of the research design is data triangulation [112], combining information from various evaluations. The benefit of this approach is that by validating various sources of verification, any conclusions are presumably to be more valid. Triangulation tests the uniformity of results obtained from different experiments thereby mitigating some of the threats influencing the results. This study uses triangulation of methods (using multiple methods to assess the same information in the context of software maintenance).

The chapter is organized in seven sections: section 5.1 presents the research questions; section 5.2 describes the evaluation methods used in the study; sections 5.3, 5.4, and 5.5 present the study of using the new coupling metric in change impact analysis, for predicting faulty classes, for predicting maintainable classes respectively; section 5.6 presents a discussion about the results of the empirical study, and section 5.7 discusses the threat to validity of the study.

5.1. Research Questions

The following research questions are addressed within the context of empirically evaluating the new coupling metric in different applications:

- RQ3: Does SSCM provide better support for ranking classes during impact analysis than other coupling metrics?

- RQ4: Does SSCM positively correlate with faults, and how much does SSCM contribute to distinguishing faulty from not faulty classes?

- RQ5: Does SSCM positively correlate with class maintainability, and how much does SSCM contribute to identifying maintainable classes?

5.2. Evaluation Methods

This section describes the evaluation methods used in this empirical study.

5.2.1. Accuracy Measures

In this study, impact analysis methods are evaluated using precision and recall. Precision represents the classes proportion recommended by a metric that are truly changed together with the given class according to bug reports. Recall represents the classes proportion that are changed together with the given class and are successfully retrieved using the coupling metrics. These measures are calculated from the approximated impact-set resulting from a technique and the actual impact-set from the actual implemented change-sets. To evaluate the performance of coupling metrics in impact analysis, available bug data are used to compute the precision and recall.

5.2.2. Correlation Analysis

Correlation analysis studies the degree to which changes in the value of an attribute (one of the coupling metrics) are associated with changes in another attribute (such as number of faults in a class). The Spearman correlation is preferred instead of Pearson correlation because the former ignores any assumptions about the data distribution [34].

5.2.3. Feature Ranking

Hall and Holmes [50] categorized feature selection algorithms as algorithms that evaluate individual attributes and algorithms that evaluate a subset of attributes. The first category of feature selection algorithms identifies which metric is able to serve as discriminatory attribute for indicating an external quality. Feature selection algorithms help in ranking and finding the most important metrics for building more robust prediction

models. In this study, since the comparison is between the new metric and existing metrics, evaluating individual attributes is chosen to see which of the coupling metrics is important in distinguishing between faulty and not faulty classes.

This study uses a feature ranking algorithm that evaluates individual attributes, ReliefF. For the evaluation of the performance of coupling metrics in identifying faulty classes and maintainable classes, ReliefF is used. Based on three different studies on feature selection techniques [9, 107, 108], ReliefF was the best feature selection method in fault-proneness models. ReliefF attribute selection [50] assesses the importance of an attribute by continually sampling an instance and considering the value of the given attribute for the nearby instance of the same and different class. The next simple example illustrates how ReliefF method works.

Here is an example to illustrate the intuition behind ReliefF. Relief algorithm is fast, easy to understand and accurate even with noisy data. The algorithm is based on a simple principle, putting objects with similar properties in a class. Some of these features are very important in the classification task and others are less important. The next simple example illustrates how Relief separates boats from cars.

Consider having many boats and cars, which have some well-known features in common, you can travel with them, they have an engine, a steering wheel etc. You can say that not every boat has an engine; some are sailing boats and hence rely on their sail for energy. However, if you ask a child what separates boats from cars, they probably say that cars have wheels and boats have not, or boats float on water and cars do not. Therefore, these are important features. This is basically what Relief does, if we pick a car, it looks for a car most similar to the one we have picked and also for a boat most similar to the car we have picked. After doing that, it looks at the features in which they differ, because these are important features for separating boats from cars.

5.3. Impact Analysis

Impact analysis involves detecting source code entities impacted by a change to a given source code entity. The coupling metrics are able to aid in ordering (ranking) classes, based on dependencies between classes, captured by the coupling metrics [26]. For more information about impact analysis please refer to Chapter 2.

5.3.1. Data Collection

A history of changes observed in the selected systems is used to determine whether the proposed coupling metric can find classes with common changes. The history data has been collected for previous versions of each project.

Each selected software project has its own bug tracking system (BTS). These BTSs have been utilized to collect the bugs and associate each bug with specific classes. After collecting the data, the data were sieved to remove bug reports that contained only one modified class. The details of the collected data are shown in Table 11.

Table 11. Details of the Collected History Changes for the Selected Projects

Project	Studied Ver.	Dates	# of Bugs	Source
POI	2.5-3.0	Feb 29, 2004-July 05,2007	621	https://issues.apache.org/bugzilla/
Synapse	1.0-1.1.1	Jun 08, 2007-Nov 18, 2010	632	https://issues.apache.org/jira/browse/SYNAPSE
Lucene	1.9-2.3.2	Feb 27, 2006-Oct 08, 2008	775	https://issues.apache.org/jira/browse/LUCENE
Ivy	2.0.0-2.0.0-rc2	Apr 26, 2007-Jan 20, 2009	489	https://issues.apache.org/jira/browse/IVY
Log4j	1.0-1.1.3	Aug 01, 2001-May 01,2002	824	https://issues.apache.org/bugzilla/
Xerces	1.2.0-1.4.3	Aug 28, 2000-Nov 15, 2001	356	https://issues.apache.org/bugzilla/

5.3.2. Empirical Comparison

Most structural coupling metrics are defined at the system level, which means they consider all dependencies for a given class c to the remainder of the classes in the system. For impact analysis, coupling metrics must be altered to consider coupling between pairs of classes only. IC, Ca, and Ce were redefined to account for class-pairs only by following the technique used in previous studies [26, 88]. IC investigates the inheritance relationship; two classes c_1 and c_2 are considered coupled if c_1 inherits from c_2. Ca investigates inbound

coupling; two classes c_1 and c_2 are considered coupled if c_1 depends on c_2. Ce investigates outbound coupling; two classes c_1 and c_2 are considered coupled if c_2 depends on c_1.

To answer the third research question (RQ3), the SSCM metric is compared with the IC, Ca, Ce, CoCC, and CCBCm metrics to assess which produces better support for impact analysis. IC, Ca, and Ce were chosen from the structural metrics as representatives since they were previously redefined to consider class-pairs [26]. CoCC and CCBCm (the maximum conceptual coupling between two classes) were chosen because a previous study [88] showed that CCBCm was the best among nine structural metrics to provide support for impact analysis. SSCM should encapsulate two aspects of coupling among classes compared to IC, Ca, Ce, CoCC, and CCBCm which utilize only structural or semantic information.

Different approaches can be utilized to find the number of candidate classes during impact analysis. The commonly used approach is to select a cut-off point inwhich a top n classes are selected from the list. For each coupling metric, the cut-off point is varied from 5, 10, and 15 classes. Average precision (P) and recall (R) for each class are computed for every metric.

The changes history can then be utilized to evaluate coupling measures in their performance in ranking of classes against actual changes in the software system. The following steps summarize the evaluation methodology:

- For a given software project, a set of bug reports is mined from the BTS. The set of modified classes to fix each bug is identified.
- For each class, pair-wise (class-to-class) coupling metrics are computed. Each metric values are utilized to computer ranks the remaining classes in the system. Each coupling metric ranks classes differently based on their coupling values to that class.

- Using a particular cut-off point, select the top n classes in each ranked list of results produced by every metric. For every class used in the evaluation, based on a specific coupling metric ranking, the effectiveness of identifying relevant classes is evaluated. Each coupling metric produces a different ranking of the coupling between the studied classes.

- To evaluate each coupling metric, the ranked lists of classes are compared to classes that were truly changed. Average precision (P) and recall (R) for each class are computed for every metric.

The results in Tables 12, 13, and 14 show that SSCM is the best indicator among the coupling metrics studied in change proneness. SSCM can be adequately utilized to order classes during impact analysis in object-oriented systems. The proposed metric performed better than the these studied metrics.

Table 12. Precision (P) and recall (R) for impact analysis based on 5 cut points

	IC		Ca		Ce		SSCM		CoCC		CCBCm	
	P	R	P	R	P	R	P	R	P	R	P	R
POI	0.07	0.06	0.06	0.05	0.08	0.07	0.26	0.12	0.11	0.01	0.18	0.9
Synapse	0.03	0.04	0.03	0.04	0.05	0.04	0.2	0.17	0.01	0.01	0.15	0.12
Lucene	0.04	0.03	0.05	0.06	0.06	0.03	0.12	0.11	0.02	0.01	0.8	0.7
Ivy	0.06	0.05	0.07	0.07	0.06	0.05	0.22	0.15	0.04	0.03	0.9	0.7
Log4j	0.06	0.05	0.07	0.08	0.09	0.07	0.185	0.11	0.143	0.02	0.155	0.08
Xerces	0.08	0.06	0.06	0.05	0.07	0.05	0.17	0.13	0.03	0.02	0.9	0.8

Table 13. Precision (P) and recall (R) for impact analysis based on 10 cut points

	IC		Ca		Ce		SSCM		CoCC		CCBCm	
	P	R	P	R	P	R	P	R	P	R	P	R
POI	0.06	0.07	0.05	0.05	0.07	0.08	0.22	0.14	0.12	0.02	0.17	0.1
Synapse	0.03	0.04	0.03	0.05	0.05	0.05	0.12	0.19	0.008	0.02	0.098	0.14
Lucene	0.03	0.04	0.04	0.06	0.05	0.04	0.1	0.12	0.011	0.01	0.07	0.08
Ivy	0.05	0.06	0.06	0.07	0.05	0.06	0.18	0.18	0.04	0.04	0.8	0.8
Log4j	0.05	0.05	0.06	0.07	0.08	0.08	0.2	0.15	0.151	0.04	0.17	0.12
Xerces	0.07	0.06	0.05	0.06	0.06	0.06	0.15	0.15	0.03	0.03	0.8	0.9

The Kruskal-Wallis test [97] is a nonparametric alternative to the one-way analysis of variance (ANOVA), which can be used when more than three independent samples are

Table 14. Precision (P) and recall (R) for impact analysis based on 15 cut points

	IC		Ca		Ce		SSCM		CoCC		CCBCm	
	P	R	P	R	P	R	P	R	P	R	P	R
POI	0.05	0.08	0.04	0.06	0.06	0.08	0.19	0.15	0.133	0.04	0.16	0.13
Synapse	0.02	0.05	0.03	0.05	0.04	0.06	0.08	0.19	0.006	0.09	0.06	0.12
Lucene	0.02	0.04	0.03	0.07	0.04	0.06	0.06	0.12	0.01	0.03	0.04	0.1
Ivy	0.04	0.07	0.05	0.06	0.04	0.07	0.14	0.22	0.04	0.05	0.7	0.8
Log4j	0.04	0.06	0.05	0.08	0.07	0.08	0.19	0.17	0.148	0.05	0.16	0.11
Xerces	0.06	0.07	0.05	0.07	0.05	0.07	0.12	0.18	0.03	0.04	0.7	0.11

Table 15. The results of two Kruskal-Wallis tests for precision and recall

	Test 1 precision	**Test 2 recall**
chi-squared	58.9	82.6
degree of freedom	5	5
p-value	< 0.001	< 0.001

present. The Kruskal-Wallis's test fits this case because it has four independent samples of precision and recall values for each coupling metric. The Kruskal-Wallis's test was executed individually for precision and recall values (see Table 15). In both tests, the p-value is very small, which indicates that the difference between the coupling metric values is statistically significant.

5.4. Fault-Proneness

Class fault-proneness can be defined as the number of faults detected in a class. Software metrics are used as independent variables and fault-proneness is the dependent variable. Several existing studies showed that coupling metrics can be used as good indicators for the fault proneness of classes in OO systems [38, 49, 89]. This study aims to investigate how much SSCM contributes to predicting fault-prone classes (RQ4). The underlying conjecture is that combining structural and semantic coupling (SSCM) should be a better indicator of coupling; hence, it is a better indicator of fault proneness.

5.4.1. Data Collection

The defect data for this study were collected by [57] and are available online at the PROMISE repository. For that study, the defects were collected using the BugInfo tool, from the selected software systems source code repositories. BugInfo analyzes the logs

of the code repositories and according to the log content decides if a commit is a bug-fix or not. These data were widely used to construct fault-proneness predictive models in the literature [82, 94, 104, 76].

5.4.2. Correlation Study

To examine the relationship between the new coupling metric and fault-proneness, an empirical study is performed with the conjecture that there is a strong correlation between structurally and semantically coupled classes and defects. The goal of this study is to demonstrate that with an increase in coupling metrics values there is a statistically significant increase in defects. Table 16 shows the Spearman rank correlation values between coupling metrics and the number of faults for each software project. With a median of 0.42, RFC shows the highest correlation of all metrics for each project. CBO and SSCM have the second strongest correlation with a median of 0.325. Table 16 shows that there is a moderate correlation between SSCM and defects.

Table 16. Spearman Correlation Between Coupling Metrics and Number of Faults

Project	DIT	CBO	RFC	Ca	Ce	NPM	IC	CBM	CoCC	SSCM
POI	0.07	0.44	0.44	0.24	0.41	0.28	0.16	0.23	0.07	0.38
Synapse	-0.05	0.3	0.45	0.29	0.25	0.23	0.08	0.09	-0.02	0.34
Lucene	0.18	0.41	0.45	0.18	0.39	0.4	0.28	0.26	0.09	0.36
Ivy	-0.07	0.34	0.4	0.05	0.15	0.29	-0.03	-0.02	0.06	0.31
Log4j	0.25	0.31	0.2	0.18	0.2	0.32	0.3	0.35	0.07	0.28
Xerces	0.12	0.23	0.32	0.14	0.25	0.34	0.18	0.17	0.03	0.26
Median	0.035	0.325	0.42	0.18	0.25	0.305	0.17	0.2	0.065	0.325

5.4.3. Feature Ranking Study

Software quality prediction channels the quality assurance effort towards the most error-prone entities. Software quality prediction models use software metrics; however, not all metrics (features) contribute the same to predicting the class attribute. Hence, choosing a small set of metrics that are more applicable to arriving at the class attribute is a critical step.

Through removing redundant attributes metrics, the fault prediction model will eventually improve. Furthermore, software practitioners prefer to work with smaller feature subsets, since they can only collect the most important metrics in future efforts. Hence, feature selection is an important component in data mining and may considerably impact the mining results.

The classification model is investigated using ReliefF and the results are presented in Table 17. ReliefF was the most robust feature ranking algorithm in fault-proneness models [9, 107, 108]. For each metric, the top two discriminant metrics are highlighted in bold in the table. The table lists the coupling metric along with their ReliefF values. The higher the value, the more importance the coupling measure has when identifying fault-prone classes. These are known as the relevant features. It has been shown that SSCM contributes to the classification in all selected systems, which indicates that it is useful in distinguishing between faulty and not faulty classes.

Table 17. Ranking of Coupling Metrics Based on ReliefF for Fault-proneness

System	DIT	CBO	RFC	Ca	Ce	NPM	IC	CBM	CoCC	SSCM
POI	0.004	0.007	0.016	0.001	0.007	0.019	0	**0.032**	0.005	**0.03**
Synapse	**0.05**	0.01	0.02	0.006	0.007	0.006	0	0.003	0.007	**0.04**
Lucene	0.01	**0.02**	0.01	0.009	0.01	0.009	0.027	0.007	0	**0.05**
Ivy	**0.13**	0.03	**0.08**	0.01	0.03	0.03	0.06	0.03	0.008	0.07
Xerces	0.2	0.05	**0.13**	0.03	0.05	0.03	0.05	**0.09**	0.02	0.08

5.5. Software Maintainability

Maintainability is “the ease with which a software component can be modified” [55]. There have been noteworthy research studies about software attributes that make software easier to maintain. These attributes include internal documentation, modularity, structured coding techniques, and code readability. A maintainability model facilitates development teams in predicting maintainability of a software system and helps in managing and planning the maintenance resources. The prediction model can also help in reducing the maintenance effort and consequently, reducing the overall project cost.

The extent of resources spent on software maintenance is much more than what is being spent on its development. Consequently, producing easy to maintain software might reduce costs and efforts. The problem of predicting software maintainability is widely acknowledged in the industry and several research studies showed how maintainability can be predicted using various processes and tools at the time of designing with the help of software design metrics [65, 31, 116, 66, 39, 32]. Studies have been conducted and found a strong link between Object Oriented software metrics and maintainability.

This case study aims to investigate if the proposed metric can enhance the performance of maintenance predictive models (RQ5). The case study shows that the proposed metric provides additional information not sufficiently provided by the related existing coupling metrics.

5.5.1. Data Collection

The class maintenance measure is considered as the quantity of LOC modified throughout the targeted maintenance history. Single line deletion, addition, or modification as a measure for maintainability was proposed by Li and Henry [65]. The number of revised LOC correlates with both maintenance effort measured in units of time [52], and maintenance cost [32]. Another reason to select revised LOCs as a measure is the fact that they are available in the reported software systems maintenance histories.

The publicly available revision repositories of the selected software systems have been used to collect the maintenance data for classes. The changes are introduced for several reasons such as fixing faults or mandatory evolutions. For each class, the version control systems of the selected projects provide the modified code for that alteration and differentiates between current and previous class versions, which includes added, changed, and deleted lines of code. For each revision, the history records all class revisions, revision ID, date, description, and revised pieces of code. The details of the collected data are shown in Table 11.

5.5.2. Correlation Study

Hypothetically, a meaningful correlation between the defined coupling metric and maintainability can be found. The Spearman rank correlation is used to analyze the correlation of coupling metrics with source code revisions (LOC).

Table 18. Spearman Correlation Between Coupling Metrics and Changed LOC

Project	DIT	CBO	RFC	Ca	Ce	NPM	IC	CBM	CoCC	SSCM
POI	0.52	0.65	0.69	0.32	0.12	0.33	0.08	0.1	0.01	0.72
Synapse	0.50	0.60	0.65	0.44	0.13	0.41	0.07	0.09	0.03	0.68
Lucene	-0.19	0.47	0.20	0.22	0.08	0.30	0.04	0.05	0.01	0.66
Ivy	-0.21	0.61	0.12	0.34	0.15	0.28	0.02	0.06	0.04	0.55
Log4j	0.28	0.60	0.45	0.36	0.14	0.12	0.02	0.03	0.01	0.58
Xerces	0.35	0.61	0.60	0.41	0.18	0.44	0.09	0.12	0.06	0.65
Median	0.315	0.61	0.525	0.35	0.13	0.315	0.055	0.075	0.02	0.66

Table 18 shows the Spearman rank correlation between coupling measures and changed LOC. With a median of 0.66, SSCM shows the highest correlation of all metrics for each project. CBO has the second strongest correlation with a median of 0.61. It still has a substantial correlation; on average, however, it is lower than SSCM. RFC comes third with a median correlation of 0.525. DIT, NPM, and Ca show a weak correlation with a median less than 0.36. Table 18 shows that there is a strong correlation between SSCM and modified LOC.

5.5.3. Feature Ranking Study

Software maintainability prediction is very helpful in focusing on refactoring and redesigning software entities that have large cost of maintenance. Software maintainability prediction models use software metrics; however, not all metrics (features) contribute the same to the class attribute (e.g., maintainable/or not). Ranking metrics that are more applicable to arriving at the class attribute is an important step in building the predictive model.

This case study labels classes based on their revision history (how many lines of code have been changed). To partition the maintainable classes into three different levels,

normalization is used. Normalization is the process of scaling individual samples to have unit norm. Three distinct class labels are assigned: low, medium, and high. Classes with relatively high numbers of revised LOC are considered as classes that have the propriety of having costly maintenance or low maintainability [52].

The next hypothesis is that the proposed coupling metric can serve as a discriminatory feature for indicating maintainability. The correlation study shows which of the coupling metrics has a significant relationship with maintainability. This study is interested in highlighting those metrics that are most helpful in increasing the certainty involved in predicting maintainability. The purpose of the ReliefF algorithm is to show how much a coupling metric help in reducing uncertainty about perceived values of maintainability.

Table 19 shows the ReliefF scores of all coupling metrics. For each metric, the top two discriminant metrics are highlighted in bold in the table. Table 19 shows that SSCM and RFC are the best predictors of the maintainability of classes. These features provide the highest scores regarding the values of the attribute.

Table 19. Ranking of Coupling Metrics Based on ReliefF for Maintainability

System	DIT	CBO	RFC	Ca	Ce	NPM	IC	CBM	CoCC	SSCM
POI	0.06	0.03	**0.10**	0	0.05	0.06	0.04	0.03	0	**0.11**
Synapse	0.05	0.02	**0.09**	0	0.04	0.02	0.03	0.04	0	**0.10**
Lucene	0.03	0.04	**0.08**	0	0.04	0.05	0.03	0.02	0	**0.07**
Ivy	0.04	0.05	**0.10**	0.01	0.04	0.07	0.05	0.04	0.02	**0.12**
Log4j	0.04	0.03	0.06	0	**0.09**	**0.07**	0.02	0.03	0	0.06
Xerces	0.02	0.03	**0.09**	0	0.04	0.01	0.04	0.05	0	**0.10**

5.6. Discussion

The new coupling metric was investigated in three different applications of software maintenance. The results of using SSCM in impact analysis showed that SSCM is the best indicator among the studied coupling metrics of classes change-proneness in object-oriented systems. This shows that SSCM can be effectively utilized during impact analysis to rank relevant classes. The new metric performed better than any of the studied metrics.

The second application is identifying faulty modules. The Spearman rank correlation showed that there is a moderate correlation between SSCM and defects. The feature ranking study showed that SSCM contributes to the classification in all selected systems, which indicates that it is useful in distinguishing between faulty and not faulty classes.

The third application is identifying maintainable classes. The Spearman rank correlation showed that there is a strong correlation between SSCM and modified LOC. The feature ranking study showed that SSCM and RFC are the best predictors of the maintainability of classes. These metrics provide the highest scores regarding the values of the attribute. To sum, SSCM demonstrated considerable practicality in three-pronged software maintenance applications.

5.7. Threats to Validity

This section discusses the main threats to the validity of this empirical study. Although only six Java open-source systems were considered in this study, the proposed metric has been shown to capture new dimensions in coupling measurement. With the aim of generalizing the results, an exhaustive evaluation is necessary that considers different domains systems, developed in different programming languages. In the comparison of coupling metrics, several structural metrics and one semantic metric were considered.

The impact analysis case study relies on the mined changed classes from bug reports patches. This might have affected the evaluation as these patches may contain inadequate information about actually changed classes. This threat is mitigated by bearing in mind only officially approved patches by module owner.

Like other change impact analysis approaches, no recommendation is given regarding selecting an optimal threshold. It is conceivable that the set of thresholds considered in this dissertation excludes the threshold that provides optimal performance. In addition, figuring out the best threshold is dependent on the specific software system. In the empirical comparison, different thresholds were considered to mitigate this threat.

A threat that is inherited from the data sets concerns the bug assignment. Like other mining software repositories approaches, faults can possibly be mapped to missing or wrong entities if entities undergo a change in signature. Likewise, automatic repository mining methods do not usually provide a complete representation of bug's history. On the other hand, because the data sets have been used and verified by other researchers [82, 94, 104], this threat is minimized.

In the fault-proneness case study, a correlation between the new coupling metric and faults is observed that may suggest that the new coupling metric can be a very good predictor of faults. However, correlation reflects goodness of fit not predictive power. In order to assess the strength of predictive power, feature ranking is performed.

In the class maintainability case study, the collected maintenance data significantly depends on the considered history of the systems. The probabilities for a class to be modified is correlated with the system age because usually faults are identified over time. Nonetheless, one of the criteria for choosing systems is the fact that the selected systems must have been actively maintained. The maintenance data that were collected for the six selected systems from the Control Version System (CVS) systems with the assumption that all maintenance revisions histories were recorded in the CVS. Despite the fact that different lines of code could slow the maintenance differently, the modified lines of code were considered similarly since it is infeasible to measure the maintenance effort for each modified line of code.

CHAPTER 6. CONCLUSIONS AND FUTURE WORK

6.1. Conclusions

The motivation of this dissertation is to support software maintenance by introducing a new coupling metric. The conjecture is that a combination of structural and semantic coupling relationships would much better explain the interaction of modules. In particular, a new coupling metric is proposed that finds jointed sets of intensely related methods in different classes by analyzing the method calls and the semantic relatedness between methods.

A new coupling metric has been introduced that captures structural and semantic information, which contributes to the area of software measurement. The new metric has been statistically and analytically validated. Through the three-pronged evaluation, the proposed metric outperformed other coupling metrics in impact analysis. Additionally, the proposed metric is a good predictor of fault-proneness. Finally, the proposed metric shows promise in predicting maintainable classes.

This dissertation makes the following contributions:

- Propose a new coupling metric that combines structural and semantic coupling.
- Statistically, analytically and empirically evaluate the new coupling metric.
- Perform three empirical studies to evaluate the new metric in the scope of software maintenance. More specifically, empirically evaluating the new metric in change impact analysis, fault-proneness prediction, and class maintainability prediction.
- The new coupling metric outperformed other coupling metrics in impact analysis.
- The new coupling metric moderately to strongly correlates with faults.
- The new coupling metric moderately to strongly correlates with maintainability.

In sum, this dissertation has introduced a new coupling metric to aid developers performing software maintenance. The new metric has been shown to be effective and beneficial and one day it could be largely embraced by maintainers and developers.

6.2. Future Work

Eventhough the work presented in this dissertation shows promising results for the new coupling metric, there is still a chance for advancement in future work. There are a number of opportunities for forthcoming efforts regarding the new coupling metric. The definition of the metric can be enhanced to utilize a finer grained approach than methods. In one of the cases studies, a statistically correlation between the new coupling metric and faults has been observed. Since correlation reflects goodness of fit not predictive power, feature ranking study was performed. Other future work involves calculating the new metric for other types of software systems to judge its generalizability.

Other avenues for future work is using the method pair coupling to prioritize test cases. Test case prioritization techniques order test cases for execution in an attempt to increase their effectiveness. Various goals are possible; one involves rate of fault detection. Improving the fault detection rate in the course of testing can accelerate feedback and enable software engineers to start correcting faults earlier. The test cases of strongly coupled methods (based on the new coupling metric) can be given a high priority due to their potential of revealing more bugs.

In addition, the class-to-class coupling metric can be utilized in re-modularizing and restructuring the software systems. Software systems are modularized to make their inherent complexity manageable. Modularization is the process of breaking a software system into a set of collaborating packages. Each of these packages should ideally have high cohesion and low coupling. The class-to-class coupling measure can be used to identify strong chains of classes. Based on these identified chains, the system can be restructured to a different packaging structure.

Moreover, software evolution is the dynamic behavior of software systems while they are maintained and improved over their lifetimes. Software systems evolve during their life cycle to accommodate new features and to improve their quality. Software needs to evolve in order to survive for a lengthy period. The changes that software undergo lie within corrective, preventive, adaptive and perfective maintenance that lead to software evolution. Most software evolution studies highlight the statistical changes of the software system by examining its evolution metrics; little effort has been carried out to understand the evolution of the software structure. The new coupling metric can be studied overtime and measure the coupling evolution behavior and its relation to defects.

REFERENCES

[1] Alain Abran, *Software metrics and software metrology*, Wiley-IEEE Computer Society Press, 2010.

[2] Mithun Acharya and Brian Robinson, *Practical change impact analysis based on static program slicing for industrial software systems*, Proceedings of the 33rd international conference on software engineering, ACM, 2011, pp. 746–755.

[3] Yunsik Ahn, Jungseok Suh, Seungryeol Kim, and Hyunsoo Kim, *The software maintenance project effort estimation model based on function points*, Journal of Software Maintenance and Evolution: Research and Practice **15** (2003), no. 2, 71–85.

[4] Fumio Akiyama, *An example of software system debugging*, Information Processing **71** (1971), 353–359.

[5] Allan J Albrecht, *Measuring application development productivity*, Proceedings of the Joint SHARE/GUIDE/IBM Application Development Symposium, vol. 10, 1979, pp. 83–92.

[6] Edward B Allen, Taghi M Khoshgoftaar, and Ye Chen, *Measuring coupling and cohesion of software modules: an information-theory approach*, Proceedings. Seventh International Software Metrics Symposium, METRICS 2001., IEEE, 2001, pp. 124–134.

[7] Mohammad Alshayeb and Wei Li, *An empirical validation of object-oriented metrics in two different iterative software processes*, IEEE Transactions on Software Engineering **29** (2003), no. 11, 1043–1049.

[8] Erik Arisholm, Lionel C Briand, and Audun Foyen, *Dynamic coupling measurement for object-oriented software*, IEEE Transactions on Software Engineering, **30** (2004), no. 8, 491–506.

[9] Erik Arisholm, Lionel C Briand, and Magnus Fuglerud, *Data mining techniques for building fault-proneness models in telecom java software*, Software Reliability, 2007. ISSRE'07. The 18th IEEE International Symposium on, IEEE, 2007, pp. 215–224.

[10] Normi Sham Awang Abu Bakar and Clive V Boughton, *Validation of measurement tools to extract metrics from open source projects*, IEEE Conference on Open Systems (ICOS), 2012, IEEE, 2012, pp. 1–6.

[11] Jagdish Bansiya and Carl G. Davis, *A hierarchical model for object-oriented design quality assessment*, IEEE Transactions on Software Engineering, **28** (2002), no. 1, 4–17.

[12] Victor R. Basili, Lionel C. Briand, and Walcelio L. Melo, *A validation of object-oriented design metrics as quality indicators*, IEEE Transactions on Software Engineering, **22** (1996), no. 10, 751–761.

[13] Gabriele Bavota, *Using structural and semantic information to support software refactoring*, Proceedings of the 2012 International Conference on Software Engineering, IEEE Press, 2012, pp. 1479–1482.

[14] Gabriele Bavota, Andrea De Lucia, Andrian Marcus, and Rocco Oliveto, *Software re-modularization based on structural and semantic metrics*, Reverse Engineering (WCRE), 2010 17th Working Conference on, IEEE, 2010, pp. 195–204.

[15] Gabriele Bavota, Bogdan Dit, Rocco Oliveto, Massimiliano Di Penta, Denys Poshyvanyk, and Andrea De Lucia, *An empirical study on the developers' perception of software coupling*, Proceedings of the 2013 International Conference on Software Engineering, IEEE Press, 2013, pp. 692–701.

[16] David Binkley and Dawn Lawrie, *Information retrieval applications in software maintenance and evolution*, Encyclopedia of Software Engineering (2009), 454–463.

[17] Shawn A Bohner, *Impact analysis in the software change process: A year 2000 perspective*, Proceedings., International Conference on Software Maintenance, 1996, IEEE, 1996, pp. 42–51.

[18] Shawn A. Bohner, *Extending software change impact analysis into cots components*, Proceedings of the 27th Annual NASA Goddard Software Engineering Workshop (SEW-27'02), IEEE Computer Society, 2002, pp. 175–182.

[19] Lionel Briand, Prem Devanbu, and Walcelio Melo, *An investigation into coupling measures for c++*, Proceedings of the 19th international conference on Software engineering, ACM, 1997, pp. 412–421.

[20] Lionel Briand, Jürgen Wüst, John Daly, and Victor Porter, *Exploring the relationship between design measures and software quality in object-oriented systems*, Journal of Systems and Software **51** (2000), 245–273.

[21] Lionel C Briand, John Daly, Victor Porter, and Jurgen Wust, *A comprehensive empirical validation of design measures for object-oriented systems*, Proceedings. Fifth International Software Metrics Symposium, Metrics 1998., IEEE, 1998, pp. 246–257.

[22] Lionel C. Briand, John W. Daly, and Jurgen K Wust, *A unified framework for coupling measurement in object-oriented systems*, IEEE Transactions on Software Engineering, **25** (1999), no. 1, 91–121.

[23] Lionel C Briand, Sandro Morasca, and Victor R Basili, *Measuring and assessing maintainability at the end of high level design*, Proceedings. IEEE International Conference on Software Maintenance, 1993., IEEE, 1993, pp. 88–87.

[24] Lionel C. Briand, Sandro Morasca, and Victor R. Basili, *Property-based software engineering measurement*, IEEE Transactions on Software Engineering, **22** (1996), no. 1, 68–86.

[25] Lionel C Briand, Jürgen Wüst, Stefan V Ikonomovski, and Hakim Lounis, *Investigating quality factors in object-oriented designs: an industrial case study*, Proceedings of the 21st international conference on Software engineering, ACM, 1999, pp. 345–354.

[26] Lionel C Briand, Jurgen Wust, and Hakim Lounis, *Using coupling measurement for impact analysis in object-oriented systems*, Proceedings. IEEE International Conference on Software Maintenance, 1999.(ICSM'99), IEEE, 1999, pp. 475–482.

[27] Jie-Cherng Chen and Sun-Jen Huang, *An empirical analysis of the impact of software development problem factors on software maintainability*, Journal of Systems and Software **82** (2009), no. 6, 981–992.

[28] Shyam R Chidamber and Chris F Kemerer, *A metrics suite for object oriented design*, IEEE Transactions on Software Engineering, **20** (1994), no. 6, 476–493.

[29] Misook Choi and JongSuk Lee, *A dynamic coupling for reusable and efficient software system*, 5th ACIS International Conference on Software Engineering Research, Management & Applications, SERA 2007., IEEE, 2007, pp. 720–726.

[30] Brendan Cleary, Chris Exton, Jim Buckley, and Michael English, *An empirical analysis of information retrieval based concept location techniques in software comprehension*, Empirical Software Engineering **14** (2009), no. 1, 93–130.

[31] Melis Dagpinar and Jens H Jahnke, *Predicting maintainability with object-oriented metrics-an empirical comparison*, Proceedings of the 10th Working Conference on Reverse Engineering (WCRE), 2003, pp. 155–164.

[32] Jehad Al Dallal, *Object-oriented class maintainability prediction using internal quality attributes*, Information and Software Technology **55** (2013), 2028–2048.

[33] Scott C. Deerwester, Susan T Dumais, Thomas K. Landauer, George W. Furnas, and Richard A. Harshman, *Indexing by latent semantic analysis*, Journal of the American Society for Information Science **41** (1990), no. 6, 391–407.

[34] Jay Devore, *Probability and statistics for engineering and the sciences*, 8th ed., Cengage Learning, Stamford, CT, 2011.

[35] Bruno Dufour, Barbara G Ryder, and Gary Sevitsky, *Blended analysis for performance understanding of framework-based applications*, Proceedings of the 2007 International Symposium on Software Testing and Analysis, ACM, 2007, pp. 118–128.

[36] Marc Eaddy, Alfred V Aho, Giuliano Antoniol, and Y-G Guéhéneuc, *Cerberus: Tracing requirements to source code using information retrieval, dynamic analysis, and program analysis*, The 16th IEEE International Conference on Program Comprehension, ICPC 2008., IEEE, 2008, pp. 53–62.

[37] Johann Eder, Gerti Kappel, and Michael Schrefl, *Coupling and cohesion in object-oriented systems*, Tech. report, University of Klagenfurt, Austria, 1994.

[38] Khaled El Emam, Walcelio Melo, and Javam C Machado, *The prediction of faulty classes using object-oriented design metrics*, Journal of Systems and Software **56** (2001), no. 1, 63–75.

[39] Mahmoud O Elish and Karim O Elish, *Application of treenet in predicting object-oriented software maintainability: A comparative study*, 13th European Conference on Software Maintenance and Reengineering, CSMR'09., IEEE, 2009, pp. 69–78.

[40] Norman Fenton, *Software measurement: A necessary scientific basis*, IEEE Transactions on Software Engineering **20** (1994), no. 3, 199–206.

[41] Norman Fenton and Austin Melton, *Deriving structurally based software measures*, Journal of Systems and Software **12** (1990), no. 3, 177–187.

[42] Norman E Fenton and Martin Neil, *Software metrics: roadmap*, Proceedings of the Conference on the Future of Software Engineering, ACM, 2000, pp. 357–370.

[43] Norman E Fenton and Shari Lawrence Pfleeger, *Software metrics: a rigorous and practical approach*, PWS Publishing Co., 1998.

[44] Harald Gall, Mehdi Jazayeri, and Jacek Krajewski, *Cvs release history data for detecting logical couplings*, Proceedings. Sixth International Workshop on Principles of Software Evolution, 2003, IEEE, 2003, pp. 13–23.

[45] Reto Geiger, Beat Fluri, Harald C Gall, and Martin Pinzger, *Relation of code clones and change couplings*, Fundamental Approaches to Software Engineering, Springer, 2006, pp. 411–425.

[46] Malcom Gethers, Bogdan Dit, Huzefa Kagdi, and Denys Poshyvanyk, *Integrated impact analysis for managing software changes*, 34th International Conference on Software Engineering (ICSE), 2012, IEEE, 2012, pp. 430–440.

[47] Malcom Gethers and Denys Poshyvanyk, *Using relational topic models to capture coupling among classes in object-oriented software systems*, IEEE International Conference on Software Maintenance (ICSM), 2010, IEEE, 2010, pp. 1–10.

[48] Tom Gilb and Gerald M Weinberg, *Software metrics*, vol. 51, Winthrop Publishers Cambridge, Massachusetts, 1977.

[49] Tibor Gyimothy, Rudolf Ferenc, and Istvan Siket, *Empirical validation of object-oriented metrics on open source software for fault prediction*, IEEE Transactions on Software Engineering, **31** (2005), no. 10, 897–910.

[50] Mark Andrew Hall and Geoffrey Holmes, *Benchmarking attribute selection techniques for discrete class data mining*, IEEE Transactions on Knowledge and Data Engineering **15** (2003), no. 6, 1437–1447.

[51] Maurice H Halstead, *Elements of software science (operating and programming systems series)*, Elsevier Science Inc., 1977.

[52] Jane Huffman Hayes, Sandip C Patel, and Liming Zhao, *A metrics-based software maintenance effort model*, Proceedings. Eighth European Conference on Software Maintenance and Reengineering, CSMR 2004, IEEE, 2004, pp. 254–258.

[53] Emily Hill, Lori Pollock, and K Vijay-Shanker, *Exploring the neighborhood with dora to expedite software maintenance*, Proceedings of the 22 IEEE/ACM International Conference on Automated Software Engineering, ACM, 2007, pp. 14–23.

[54] Martin Hitz and Behzad Montazeri, *Measuring coupling and cohesion in object-oriented systems*, Proceedings of the International Symposium on Applied Corporate Computing, vol. 50, 1995, pp. 75–76.

[55] ISO/IEC/IEEE, *Systems and software engineering - Vocabulary, 24765*, Tech. report, IEEE, 2010.

[56] Hsin-Yi Jiang, Tien N Nguyen, Xiang Chen, Hojun Jaygarl, and Carl K Chang, *Incremental latent semantic indexing for automatic traceability link evolution management*, 23rd IEEE/ACM International Conference on Automated Software Engineering, ASE 2008, IEEE, 2008, pp. 59–68.

[57] Marian Jureczko and Diomidis Spinellis, *Using object-oriented design metrics to predict software defects*, Models and Methods of System Dependability. Oficyna Wydawnicza Politechniki Wrocławskiej (2010), 69–81.

[58] Sadaf Khalid, Saima Zehra, and Fahim Arif, *Analysis of object oriented complexity and testability using object oriented design metrics*, Proceedings of the 2010 National Software Engineering Conference, ACM, 2010, p. 4.

[59] Barbara Kitchenham, Shari Lawrence Pfleeger, and Norman Fenton, *Towards a framework for software measurement validation*, IEEE Transactions on Software Engineering, **21** (1995), no. 12, 929–944.

[60] Barbara A. Kitchenham and John G. Stell, *The danger of using axioms in software metrics*, IEE Proceedings-[see also Software, IEE Proceedings] Software Engineering, vol. 144, IET, 1997, pp. 279–285.

[61] Lawrence A Klein, *Sensor and data fusion: a tool for information assessment and decision making*, vol. 324, SPIE Publications, Bellingham, WA, 2004.

[62] Stefan Kramer and Hermann Kaindl, *Coupling and cohesion metrics for knowledge-based systems using frames and rules*, ACM Transactions on Software Engineering and Methodology (TOSEM) **13** (2004), no. 3, 332–358.

[63] Adrian Kuhn, Stéphane Ducasse, and Tudor Gírba, *Semantic clustering: Identifying topics in source code*, Information and Software Technology **49** (2007), no. 3, 230–243.

[64] YS Lee, BS Liang, SF Wu, and FJ Wang, *Measuring the coupling and cohesion of an object-oriented program based on information flow*, Proc. International Conference on Software Quality, Maribor, Slovenia, 1995, pp. 81–90.

[65] Wei Li and Sallie Henry, *Object-oriented metrics that predict maintainability*, Journal of Systems and Software **23** (1993), no. 2, 111–122.

[66] Wang Li-Jin, Hu Xin-Xin, Ning Zheng-Yuan, and Ke Wen-hua, *Predicting object-oriented software maintainability using projection pursuit regression*, 1st International Conference on Information Science and Engineering (ICISE), 2009, IEEE, 2009, pp. 3827–3830.

[67] Jonathan I Maletic and Andrian Marcus, *Supporting program comprehension using semantic and structural information*, Proceedings of the 23rd International Conference on Software Engineering, IEEE Computer Society, 2001, pp. 103–112.

[68] Andrian Marcus and Jonathan I Maletic, *Identification of high-level concept clones in source code*, Proceedings. 16th Annual International Conference on Automated Software Engineering, ASE 2001, IEEE, 2001, pp. 107–114.

[69] Andrian Marcus, Jonathan I Maletic, and Andrey Sergeyev, *Recovery of traceability links between software documentation and source code*, International Journal of Software Engineering and Knowledge Engineering **15** (2005), no. 05, 811–836.

[70] Andrian Marcus, Denys Poshyvanyk, and Rudolf Ferenc, *Using the conceptual cohesion of classes for fault prediction in object-oriented systems*, IEEE Transactions on Software Engineering, **34** (2008), no. 2, 287–300.

[71] Andrian Marcus, Andrey Sergeyev, Vaclav Rajlich, and Jonathan I Maletic, *An information retrieval approach to concept location in source code*, Proceedings. 11th Working Conference on Reverse Engineering, 2004, IEEE, 2004, pp. 214–223.

[72] Robert Martin, *Oo design quality metrics - an analysis of dependencies*, Workshop Pragmatic and Theoretical Directions in Object-Oriented Software Metrics. OOPSLA'94, 1994.

[73] Thomas J McCabe, *A complexity measure*, IEEE Transactions on Software Engineering, (1976), no. 4, 308–320.

[74] GJ Myers, *Composite/structured design.*, Van Nostrand Reinhold, 1978.

[75] A Jefferson Offutt, Mary Jean Harrold, and Priyadarshan Kolte, *A software metric system for module coupling*, Journal of Systems and Software **20** (1993), no. 3, 295–308.

[76] Ahmet Okutan and Olcay Taner Yıldız, *Software defect prediction using bayesian networks*, Empirical Software Engineering **19** (2014), no. 1, 154–181.

[77] Hector M Olague, Letha H Etzkorn, Sampson Gholston, and Stephen Quattlebaum, *Empirical validation of three software metrics suites to predict fault-proneness of object-oriented classes developed using highly iterative or agile software development processes*, IEEE Transactions on Software Engineering **33** (2007), no. 6, 402–419.

[78] Anthony M Orme, H Tao, and Letha H Etzkorn, *Coupling metrics for ontology-based system*, IEEE Software **23** (2006), no. 2, 102–108.

[79] Alessandro Orso, Taweesup Apiwattanapong, and Mary Jean Harrold, *Leveraging field data for impact analysis and regression testing*, ACM SIGSOFT Software Engineering Notes **28** (2003), no. 5, 128–137.

[80] Ganesh J Pai and Joanne Bechta Dugan, *Empirical analysis of software fault content and fault proneness using bayesian methods*, IEEE Transactions on Software Engineering, **33** (2007), no. 10, 675–686.

[81] David Lorge Parnas, *Designing software for ease of extension and contraction*, IEEE Transactions on Software Engineering, (1979), no. 2, 128–138.

[82] Fayola Peters, Tim Menzies, and Andrian Marcus, *Better cross company defect prediction*, Proceedings of the Tenth International Workshop on Mining Software Repositories, IEEE Press, 2013, pp. 409–418.

[83] Maksym Petrenko and Václav Rajlich, *Variable granularity for improving precision of impact analysis*, IEEE 17th International Conference on Program Comprehension, ICPC'09., IEEE, 2009, pp. 10–19.

[84] M Piattini, C Calero, M Polo, and F Ruiz, *Maintainability in object-relational databases*, Proceedings of the 1st European Software Measurement Conference (FESMA98), 1998, pp. 223–230.

[85] Geert Poels and Guido Dedene, *Distance-based software measurement: necessary and sufficient properties for software measures*, Information and Software Technology **42** (2000), no. 1, 35–46.

[86] Denys Poshyvanyk, Y-G Guéhéneuc, Andrian Marcus, Giuliano Antoniol, and Vaclav Rajlich, *Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval*, IEEE Transactions on Software Engineering **33** (2007), no. 6, 420–432.

[87] Denys Poshyvanyk and Andrian Marcus, *The conceptual coupling metrics for object-oriented systems*, 22nd IEEE International Conference on Software Maintenance, ICSM'06., IEEE, 2006, pp. 469–478.

[88] Denys Poshyvanyk, Andrian Marcus, Rudolf Ferenc, and Tibor Gyimóthy, *Using information retrieval based coupling measures for impact analysis*, Empirical Software Engineering **14** (2009), no. 1, 5–32.

[89] Danijel Radjenovic, M Herico, Richard Torkar, and Aleš Zivkovic, *Software fault prediction metrics: A systematic literature review*, Information and Software Technology **55** (2013), 1397–1418.

[90] Vaclav Rajlich and Prashant Gosavi, *Incremental change in object-oriented programming*, IEEE Software **21** (2004), no. 4, 62–69.

[91] Meghan Revelle, Malcom Gethers, and Denys Poshyvanyk, *Using structural and textual information to capture feature coupling in object-oriented software*, Empirical Software Engineering **16** (2011), no. 6, 773–811.

[92] Meghan Revelle and Denys Poshyvanyk, *An exploratory study on assessing feature location techniques*, IEEE 17th International Conference on Program Comprehension, ICPC'09, IEEE, 2009, pp. 218–222.

[93] Pablo Rossi and George Fernandez, *Definition and validation of design metrics for distributed applications*, Proceedings. Ninth International Software Metrics Symposium, 2003, IEEE, 2003, pp. 124–132.

[94] Giuseppe Scanniello, Carmine Gravino, Andrian Marcus, and Tim Menzies, *Class level fault prediction using software clustering*, Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering (ASE2013), IEEE, 2013.

[95] Norman F. Schneidewind, *Methodology for validating software metrics*, IEEE Transactions on Software Engineering **18** (1992), no. 5, 410–422.

[96] Hidetoshi Shimodaira, *Approximately unbiased tests of regions using multistep-multiscale bootstrap resampling*, The Annals of Statistics **32** (2004), no. 6, 2616–2641.

[97] Peter Sprent and Nigel C Smeeton, *Applied nonparametric statistical methods*, 4th ed., Chapman & Hall/CRC Texts in Statistical Science, CRC Press, Boca Raton, FL, 2007.

[98] Ramanath Subramanyam and Mayuram S. Krishnan, *Empirical analysis of ck metrics for object-oriented design complexity: Implications for software defects*, IEEE Transactions on Software Engineering, **29** (2003), no. 4, 297–310.

[99] Robert Tairas and Jeff Gray, *An information retrieval process to aid in the analysis of code clones*, Empirical Software Engineering **14** (2009), no. 1, 33–56.

[100] Mei-Huei Tang, Ming-Hung Kao, and Mei-Hwa Chen, *An empirical study on object-oriented metrics*, Proceedings. Sixth International Software Metrics Symposium, 1999, IEEE, 1999, pp. 242–249.

[101] Jean Tessier, *Dependency finder*, `http://depfind.sourceforge.net/, accessed February, 2013.`

[102] Paolo Tonella, *Concept analysis for module restructuring*, IEEE Transactions on Software Engineering, **27** (2001), no. 4, 351–363.

[103] Stéphane Tufféry, *Data mining and statistics for decision making*, 2nd ed., Wiley Series in Computational Statistics, John Wiley & Sons, 2011.

[104] Burak Turhan, Ayşe Tosun Mısırlı, and Ayşe Bener, *Empirical evaluation of the effects of mixed project data on learning defect predictors*, Information and Software Technology (2012), 1101–1118.

[105] Béla Újházi, Rudolf Ferenc, Denys Poshyvanyk, and Tibor Gyimóthy, *New conceptual coupling and cohesion metrics for object-oriented systems*, 10th IEEE Working Conference on Source Code Analysis and Manipulation (SCAM), 2010, IEEE, 2010, pp. 33–42.

[106] Yair Wand and Ron Weber, *An ontological model of an information system*, IEEE Transactions on Software Engineering, **16** (1990), no. 11, 1282–1292.

[107] Huanjing Wang, Taghi M Khoshgoftaar, and Randall Wald, *Measuring robustness of feature selection techniques on software engineering datasets*, IEEE International Conference on Information Reuse and Integration (IRI), 2011, IEEE, 2011, pp. 309–314.

[108] Huanjing Wang, Taghi M Khoshgoftaar, Randall Wald, and Amri Napolitano, *A comparative study on the stability of software metric selection techniques*, 11th International Conference on Machine Learning and Applications (ICMLA), 2012, vol. 2, IEEE, 2012, pp. 301–307.

[109] Xiaoyin Wang, Lu Zhang, Tao Xie, John Anvik, and Jiasu Sun, *An approach to detecting duplicate bug reports using natural language and execution information*, Proceedings of the 30th International Conference on Software Engineering, ACM, 2008, pp. 461–470.

[110] MP Ware, F George Wilkie, and Mary Shapcott, *The application of product measures in directing software maintenance activity*, Journal of Software Maintenance and Evolution: Research and Practice **19** (2007), no. 2, 133–154.

[111] F George Wilkie and Barbara A Kitchenham, *Coupling measures and change ripples in c++ application software*, Journal of Systems and Software **52** (2000), no. 2, 157–164.

[112] Robert K Yin, *Applications of case study research*, 3rd ed., Applied Social Research Methods, Sage, Thousand Oaks, CA, 2011.

[113] Edward Yourdon and Larry L Constantine, *Structured design: fundamentals of a discipline of computer program and systems design*, Prentice-Hall, Inc., 1979.

[114] Ping Yu, Tarja Systa, and Hausi Muller, *Predicting fault-proneness using oo metrics. an industrial case study*, Software Maintenance and Reengineering, 2002. Proceedings. Sixth European Conference on, IEEE, 2002, pp. 99–107.

[115] Jianjun Zhao, *Measuring coupling in aspect-oriented systems*, Proceedings. of 10th IEEE International Soft. Metrics Symposium (METRICS'04), 2004.

[116] Yuming Zhou and Hareton Leung, *Predicting object-oriented software maintainability using multivariate adaptive regression splines*, Journal of Systems and Software **80** (2007), no. 8, 1349–1361.

[117] Horst Zuse, *A framework of software measurement*, Walter de Gruyter, Hawthorne, NJ, 1998.