AN EMPIRICAL STUDY ON OBJECT-ORIENTED SOFTWARE DEPENDENCIES: LOGICAL, STRUCTURAL AND SEMANTIC

A Thesis submitted as partial fulfillment of the requirement of Doctor of Philosophy (Ph.D.)

by

Nemitari Miebaka Ajienka

Computer Science Department

Brunel University London

26 January 2018

Abstract

Three of the most widely studied software dependency types are the structural, logical and semantic dependencies. Logical dependencies capture the degree of co-change between software artifacts. Semantic dependencies capture the degree to which artifacts, comments and names are related. Structural dependencies capture the dependencies in the source code of artifacts.

Prior studies show that a combination of dependency analysis (e.g., semantic and logical analysis) improves accuracy when predicting which artifacts are likely to be impacted by ripple effects of software changes (though not to a large extent) compared to individual approaches. In addition, some dependencies could be hidden dependencies when an analysis of one dependency type (e.g., logical) does not reveal artifacts only linked by another dependency type (semantic). While previous studies have focused on combining dependency information with minimal benefits, this Thesis explores the consistency of these measurements, and whether hidden dependencies arise between artifacts, and in any of the axes studied.

In this Thesis, 79 Java projects are empirically studied to investigate (i) the direct influence and the degree of overlap between dependency types on three *axes* (logical – structural (LSt); logical – semantic (LSe); structural – semantic (StSe)) (structural, logical and semantic), and (ii) the presence of hidden coupling on the *axes*. The results show that a high proportion of hidden dependencies can be detected on the LSt and StSe axes. Notwithstanding, the LSe axis shows a much smaller proportion of hidden dependencies. Practicable refactoring methods to mitigate hidden dependencies are proposed in the Thesis and discussed with examples.

Contents

List of Figures

List of Tables

Acronyms

OSS Open-Source Software

CIA Change Impact Analysis

IR Information Retrieval

IDE Integrated Development Environment

HD Hidden Dependencies

OO Object-Oriented

FOSS Free and Open-Source Software

CBO Coupling Between Objects

LOC Lines of Code

SE Software Engineering

CBSD Component Based Software Development

MSR Minning Software Repositories

API Application Programming Interface

CVS Concurrent Versioning System

NLP Natural Language Processing

LSt Logical-Structural coupling axis

LSe Logical-Semantic coupling axis

StSe Structural-Semantic coupling axis

Acknowledgements

I would like to thank my supervisor, Dr. Andrea Capiluppi for his advice, mentorship, unconditional support and encouragement throughout my studies. Our discussions always gave me the confidence of being on the right path. I also owe special thanks to Professor Steve Counsell my second supervisor, for his encouraging comments and suggestions throughout this research. I cannot forget to thank other academics like Professor Tracy Hall, Professor Martin Shepperd and Professor Rob Hierons who encouraged me during my studies.

Lastly I owe a special thanks to Ela Heaney, members of my academic review panel (Dr. Larisa Soldatova, and Dr. Yongmin Li) and all my colleagues in the Department of Computer Science whose support kept me going.

Dedication

This thesis is dedicated with endless love and gratitude to: my parents Professor Joseph and Mrs Mercy Ajienka who have been always supportive, encouraging (despite the distance), patient and worked hard to see that I succeed in all my endeavours, to my siblings Soala Ajienka and Orabelema Anene whose prayers and jokes brought me happiness and cheerfulness during the completion of my studies, to my partner Cynthia Ola for her tolerance, patience and love, to my niece Lolia Anene whose birth brought me joy during the completion of this research and lastly to my grandmothers whose prayers helped me persevere.

Chapter 1

Introduction

Software systems increasingly and unavoidably evolve to keep up with user requirements and maintenance needs [152, 197]. Small program or source code changes can have unexpected and non-local ripple effects in software systems [102] and this can complicate software maintenance. For example, adding new functionality to a component or class in object-oriented (OO) software (composed of inter-dependent objects with states and behaviours) can affect the behaviour of other essential (and directly dependent) features throughout the software [170]. As a result of the non-locality of the impact of change, it is important to ensure that the process of predicting or identifying the classes that might be impacted by a given change request (the change impact set) is as efficient or as precise as possible. In other words, the estimated impact set should mirror the actual change impact set as much as possible [1, 103, 114]. This process is known as change impact analysis (CIA).

1.1 Change impact analysis

The goal of Change Impact Analysis (CIA) is to reveal potential change impact sets [102, 159]: software artifacts that might be affected by given change requests [39]. Techniques and tools have been designed based on static structural [39] and logical software dependencies [102, 209] to identify artifacts that make up change impact sets and to predict further changes that will have to be made when an artifact is changed. For example, when a software programmer makes a change to a software artifact, a tool might help suggest further artifacts that will probably need modifications. According to Zimmermann *et al.* [210], the tool ROSE correctly predicts 26% of further files to be changed based on historical co-evolution data. Briand *et al.* [34] in their study on CIA adopted the static coupling between objects (CBO) metric introduced by Chidamber and Kemerer [46] as a binary indicator (0 or 1) of whether two classes are coupled or not, when one class uses the functionality of another class.

When maintaining OO software systems, developers need to deal with various inter-dependencies between classes or artifacts [34, 40, 104]. Studies in the area of software maintenance and evolution have introduced novel dependency identification techniques that expose new and "Hidden" Dependencies (HD) [190, 205] between classes, sometimes not captured by source code (structural dependencies) and also assessed differently from structural dependency assessment. As an example of a hidden dependency, File A could be structurally unaware of the existence of File B in a software system (i.e., when they share no structural dependency), but it frequently co-evolves with File B [205]. A study by Briand *et al.* [34] revealed that

> 'if developers are required to handle a large set of dependencies, they would omit a significant number of them'.

1.2 Dimensions of software coupling

Coupling is a measure of the interdependence between software artifacts [34]. Software engineering researchers have proposed tools and techniques to predict where change impact sets will arise. Most approaches rely on the software coupling information derived from static analysis [1, 102].

Various coupling measures have been proposed over the years, such as, *dynamic* coupling (i.e., call relationships between classes during program execution), *structural* coupling (i.e., structural relationships between classes, such as the number of method calls between them) [187], *logical* coupling (i.e., the use of historical data to identify classes that always co-change) [66] and *semantic* coupling (i.e., the degree to which identifiers and comments from classes relate to each other) [155]. While structural and semantic dependencies play major roles in software evolution, their relationship has not been empirically investigated in a large-scale empirical study.

Dynamic coupling is centred around call relationships between software artifacts during program execution (when the software is in use) [11]. Notwithstanding, investigating dynamic coupling is expensive [12] because of the processes involved, i.e., extracting data during program execution as well as analyzing the data after execution. Some researchers have tried to come up with static techniques [120] for studying dynamic coupling as opposed to dynamic methods. The complexities involved in analyzing dynamic coupling has led to the other three static forms of coupling (*logical, structural and semantic*) becoming widely adopted in software maintenance and evolution research.

Logical coupling refers to evolutionary or change dependencies that are established among software artifacts that are frequently changed together (although not necessarily structurally related) [102, 148]. It is a measure of the degree to which software artifacts co-change. This information is derived from historical data [9, 66] by analyzing patterns, relationships and relevant information of source code changes mined from multiple versions of software systems in software repositories (e.g., GitHub and Subversion) with appropriate repository mining tools (e.g., CVSAnaly [26, 168, 169]). The research domain concerned with extracting information from software repositories is widely known as *mining software repositories (MSR)* [88].

Structural coupling (also known as syntactic dependencies) [148] occurs whenever a compilation unit depends on another at compilation or linkage time. Adapting the definition of coupling as presented in [71]: "There is a directed dependency between software components A and B if A depends on B in such a way that A is not operational without B". In the case of the Java programming language, this means that A would not compile[1] in the absence of B.

In the last few years, a new dimension has been identified as an implicit or hidden dependency (HD) type, termed "*Semantic*" coupling [102, 155, 156]. Researchers have speculated that this coupling type could have an influence on structural and logical coupling but this has not been empirically investigated in prior research [148]. Simply expressed, semantic coupling is a measure of how loosely or closely related two software artifacts are, by considering the semantic information embedded in their comments and identifiers. According to Bavota *et al.* [21]:

> 'the peculiarity of the semantic coupling measure allows it to better estimate the mental model of developers than the other coupling measures (i.e., structural or logical). This is because, in several cases, the interactions between classes are encapsulated in the source code vocabulary (...)'.

1.3 Coupling axes

In Section 1.2, we introduced the different dimensions of software coupling - dynamic and static (structural, logical and semantic) software coupling. The different aforementioned forms of static software coupling (structural, logical and semantic) capture different and interesting aspects of coupling between software artifacts and these have previously been combined to derive metrics to support various software maintenance and evolution activities such as program comprehension to understand software links [155], change impact analysis (CIA) [75, 102, 103, 145, 163], refactoring, fault-proneness [7, 163], and change prediction [89, 126, 185, 202].

These coupling dimension combinations have formed three main coupling axes; LSe (logical $\leftrightarrow$ semantic), StSe (structural $\leftrightarrow$ semantic) and LSt (logical $\leftrightarrow$ struc-

[1]Transforming high-level computer programs into machine understandable instructions for execution.

tural). Figure 1.1 depicts the three different axes composed of the coupling dimensions and studied in this Thesis.

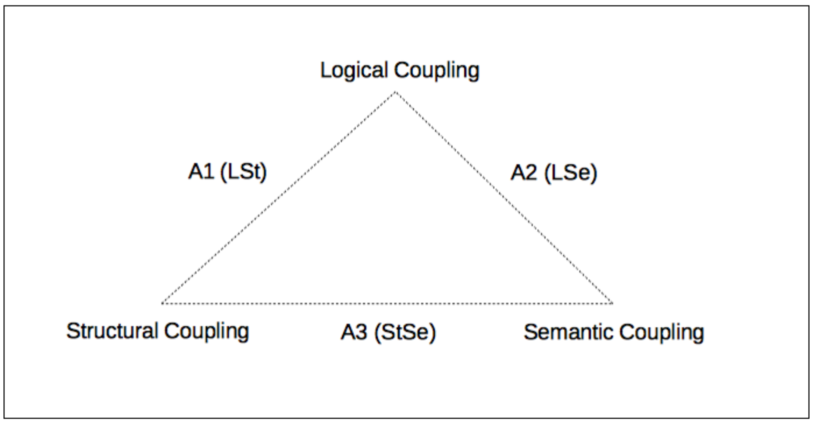


Figure 1.1: The three explored coupling axes in this Thesis based on identified gaps in the literature (see Chapter 2)

On the logical and structural coupling axis, researchers have studied how different types of structural relationships between classes influence change propagation [147]. Others have performed cross-project change prediction using open source projects [126]. On the logical and semantic coupling axis studies have been conducted on combining semantic and logical couplings to support change impact analysis in source code [102, 103]. Lastly, on the structural and semantic coupling axis, structural and semantic information have been combined to capture feature coupling in OO software [163]. Researchers have also performed studies to compare semantic coupling with structural coupling metrics [75, 155], carry out semantic decoupling[2] [145], capture method level coupling with a combination of structural and textual information in object-oriented software [163], and also developed new coupling metrics with a combination of structural and semantic coupling [7].

[2]reducing the impact of requirement changes

1.4 Hidden coupling

Kagdi and Maletic have concluded that there is a *hidden dependency* (HD) between two classes or two methods if the classes or the methods are changed at the same time in the past [100]. As Yu *et al.* stated [205]: '*hidden dependencies among software artefacts make both understanding and maintenance difficult*'.

In a similar way to logical coupling, complex dependencies are captured by semantic information which is hard to detect by traditional program analysis techniques [190]. The motivation behind logical and semantic coupling is to capture dependencies not derived from analysing structural coupling [21].

Petrenko and Rajlich state that "some CIA tools do not discover hidden dependencies, and it is the responsibility of the programmer to correctly identify and trace hidden dependencies during change impact analysis" [154].

Hidden dependencies arise in the three axes described in Section 1.3 when one dependency type between class pairs is not mirrored by another dependency type. With reference to the coupling axes described in Section 1.3, there are three main scenarios in Table 1.1, which are as follows:

- The class pairs are linked by both types of coupling along the axis (e.g., structurally and semantically linked classes). In this case there is an overlap of coupling types and there are no hidden dependencies. An analysis of either structural or semantic coupling captures both classes as being linked.
- The class pairs are only linked by one out of the two coupling types along the axis (e.g., ONLY structurally or semantically or logically linked classes). In this case there is a hidden dependency depending on the type of coupling analysis performed (e.g., an analysis of structural dependencies not noticing a semantic coupling).
- The class pairs do not share any link.

As an example, the contingency Table 1.1 shows the possible occurrences along the structural and semantic coupling axis.

When pairs of classes are linked both structurally and semantically, we posit that a dependency is established (denoted ‘E’). If a structural link is present, but not a semantic one, there could be a strong (denoted ‘S’) missing dependency. On the contrary, if there is a semantic link, but not the structural one, a weak (denoted ‘W’) missing dependency is detected. When neither a semantic nor structural link is detected, no dependency (denoted ‘x’) is established. If the structural and semantic class dependency types are established, the precision achieved when predicting ripple effects of changes can be improved. On the other hand, when dependencies are hidden or weak, developers will detect a smaller number of dependencies capable of propagating further change.

The described notion (E, S, W and x) is a novel contribution to the state of the art in software coupling. This notion and Table 1.1 will be adopted when demonstrating the proportion of unnoticed class dependencies defined along the other two studied axes[3] in the rest of the Thesis.

For example when there is no co-change history for a pair of classes but these classes share a structural or semantic link between them, an analysis of co-evolution or logical coupling will not reveal such connected class pairs. The same applies when there is a semantic link but not a structural one, and an analysis of structural coupling does not reveal the class pair because they share only a semantic link. In this Thesis, we refer to the unnoticed dependencies as **hidden**.

Table 1.1: Hidden dependencies example

Structural vs Semantic Dependencies

	semantic	*not semantic*
structural	E	S
not structural	W	x

[3]The logical and structural coupling axis, and the logical and semantic coupling axis.

1.4.1 Links between hidden coupling

Previous research [71, 148] has identified that a majority of structural dependencies are not linked to logical coupling. Among the aims of this Thesis is to investigate the link between logical and semantic coupling to identify their direct influence on each other.

A potential finding is the likelihood of a proportion of the hidden dependencies in the logical and structural axis (class pairs with a logical but not a structural link) being linked with those in the logical and semantic coupling axis (class pairs with a logical but not a semantic link) because both structural and semantic dependencies are propagators of ripple effects of change [104, 126]. The interplay between the three coupling types logical, structural and semantic based on prior work is introduced in Section 1.5.

1.5 Interplay on the coupling axes

In this Thesis, the term *interplay* refers to the direct influence of pairs of coupling dimensions described in Section 1.2 over each other along the three coupling axes in Figure 1.1.

The interplay between the widely explored coupling dimensions (logical, structural and semantic coupling) described in Section 1.2 has not been extensively studied and the degree of their relationship in OO software systems remains vague [148]. This problems form the core of this Thesis. Thus, the focus of this Thesis is to empirically investigate the interplay between the three widely studied static OO software dependencies; the structural, logical and semantic coupling at the class (or file) level of granularity. This will highlight the extent to which the coupling types overlap and the proportion of **hidden** dependencies that need refactoring [190].

We expect that the empirical investigations carried out in this Thesis will reveal a *large* overlap between semantic and logical dependencies but a *smaller* overlap between structural and semantic dependencies or logical dependencies. This is because, prior research by Bavota *et al.* showed that in several cases, the interactions between classes are encapsulated in the source code comments and identifiers of classes and methods [21] and not explicit structural relationships. Other researchers have identified that semantic dependencies are change propagators [103].

Researchers have laid emphasis on the need to study the interplay between the semantic and structural coupling as well as between semantic and logical coupling. Logical coupling has been studied in relation to structural coupling [71, 147, 148, 203] and software quality [54, 210]. However, only one study has investigated the linear relationship between degrees of structural and logical coupling and on a limited software sample [203].

Results from prior research has shown that, in general, it is more likely that two OO software classes will not co-change just because one structurally depends on the other. However, the rate with which a class co-changes with another is higher when the former structurally depends on the latter [147]. On the logical or change coupling side, prior research has reported several cases where software changes could not be justified using structural dependencies. This means that co-changes are possibly induced by other indirect kinds of software relationships (e.g., semantic coupling) [71, 148].

Geipel and Schweitzer [71] stated that “the question about the causes of change propagation has been skipped in many studies on change prediction [89, 126, 185, 202, 210] in favour of a predictive approach in which these causes are implicitly contained in a prediction function or as inputs to a machine learning algorithm". For these reasons, it is important to empirically investigate the interplay between the trio of static software coupling types: logical, structural and semantic coupling in depth. In Section 2.5 of Chapter 2, we describe the interplay between the coupling dimensions along the three coupling axes: LSt, LSe and StSe.

1.6 Research apparatus

The overarching aim of this Thesis is to synthesize the interplay between the trio of widely used coupling measurements (structural, logical and semantic) among classes in object-oriented software.

Research in the area of change impact analysis (CIA) has identified that static structural coupling metrics are generally associated with change-proneness and an advantage of studying coupling measures in relation to change propagation is that they are inherently related to ripple effects since common changes are usually due to relationships between classes [174] in OO software. However, little focus has been placed on how frequent pairs of classes are changed together in relation to semantic as well as the relationship between software architecture and semantic coupling. These are investigated in this Thesis with a sample of 79 different OSS projects used in a variety of domains (games, music, teaching, etc.).

Hypotheses help to steer the direction of research as well as predict expected outcomes [144]. For each research objective, we define related research questions and hypothesis as summarized in Table 1.2. Related statistical tests are summarized in Table 1.3 and elaborated upon in Chapter 3.

1.6.1 Research aims and objectives

Following the interplay along the three coupling axes that compose the empirical studies in this Thesis as outlined in Section 1.5, the following are the detailed **objectives** of this Thesis:

Obj1 [**To investigate the interplay between structural and logical software dependencies**] The objective is to evaluate the impact of structural and logical class dependencies on each other. This will be achieved by investigating the presence of a linear relationship between the strengths of both types of coupling between classes and statistically analyzing the overlap of the coupling types between class pairs to detect the proportion of **hidden**

dependencies and propose viable refactoring techniques.

Obj2 [**To investigate the interplay between logical and semantic software dependencies**] The objective is to evaluate the impact of semantic and logical class dependencies on each other. This will be achieved by investigating the presence of a linear relationship between the strengths of both types of coupling between classes and statistically analyzing the overlap of the coupling types between class pairs to detect the proportion of **hidden** dependencies and propose viable refactoring techniques.

Obj3 [**To investigate the interplay between structural and semantic software dependencies**] The objective is to evaluate the impact of structural and semantic class dependencies on each other. This will be achieved by investigating the presence of a linear relationship between the strengths of both types of coupling between classes and statistically analyzing the overlap of the coupling types.

Solutions to maximize the overlap between structural and semantic dependencies will also be proposed. To improve testing efforts, as well as computational efficiency and detection of **hidden** dependencies during class dependency-based change impact analysis in OO software.

Rationale: In relation to **Obj1, Obj2 and Obj3**, establishing whether there is an interplay and a large overlap between the three forms of software dependencies (structural, logical and semantic) at the class level of granularity in OO software has several **applications** in the software engineering and maintenance domain, including:

1. **Prediction of software changes**: Geipel and Schweitzer [71] state that the question about the causes of change propagation has been overlooked by many researchers in favour of a predictive approach. As such, these causes are implicitly contained in a prediction function or as input to a

machine learning algorithm in change prediction studies [89, 126, 185, 202, 210]. A strong relationship between logical and structural as well as semantic coupling provides statistical support for these models and predictions, thus helping to achieve more focused software maintenance.

2. **Focused refactoring effort**: if an analysis of structural coupling reveals a system designed with low code coupling between classes; when the analysis of the change history of classes in a software system revealed a high degree of logical coupling between classes, this will be an indication of possible targets for restructuring to decrease unnoticeable coupling between classes in the system [203]. The same scenario applies to the interplay between structural and semantic coupling.

3. **Focused testing effort**: the relationship between structural and semantic coupling to change coupling or ripple effects [1, 4, 102, 156] cannot be overestimated. When changes are made to one class, other classes with strong structural or semantic coupling to that class also need to be tested. This is to ensure that the changes in one class do not introduce regression faults in other classes. If the overlap between structural and semantic dependencies is large, then these tests only have to be carried out once. Either ONLY semantically coupled class pairs or ONLY structurally coupled pairs will need to be tested.

4. **Architecture reconstruction and monitoring**: A strong link between semantic and structural coupling will help to monitor and update the software architecture as well as reconstruct it. Research by Dragomir *et al.* shows that existing tools for architecture reconstruction and monitoring still lack the needed “ingredients” for reconstructing software architecture in the absence of documentation to support the understanding of a software [56]. Findings from the structural and semantic coupling axis provide applicable knowledge for architecture reconstruction and monitoring tools.

1.6.2 Research questions

Research **questions** were derived from each research objective in Section 1, and testable **hypotheses** formulated, as summarised in Table 1.2. An overview of statistical tests adopted in various parts of the Thesis to test the hypotheses are outlined in Table 1.3 and further explained in Chapter 3.

1.6.3 Research contributions and beneficiaries

The contributions of this thesis are six-fold[5], and can be presented as follows:

C1 – An interplay between software dependencies. This Thesis has probed and synthesized the interplay between three types of software dependencies (structural, logical and semantic coupling). This contribution adds knowledge to the state of the art in the software engineering literature.

C2 – The role of hidden dependencies. This Thesis reveals the percentage of hidden dependencies in OO software. This is based on an analysis of any one of the three types of static coupling (logical, structural and semantic) investigated.

C3 – Refactoring the hidden dependencies. Refactoring approaches are proposed where hidden dependencies occur between structural and semantic coupling. The overall aims of these approaches are to reduce testing and maintenance efforts.

C4 – Prediction of software changes. Using structural coupling information to predict unplanned future co-changes of classes is a realistic objective [147]. But we contribute that this objective is realistic with the support of semantic coupling metrics. The rationale is the small overlap between structural and logical dependencies as well as between semantic and logical coupling.

[5]The contributions and beneficiaries of this Thesis are described in detail in Chapter 7.

Table 1.2: Summary of research questions

Research Objective	Research Questions	Null Hypotheses H_0	Alternative Hypotheses H_0
Obj1	Is there a large overlap between structural and logical coupling? Is there a linear relationship between structural and logical class dependencies?	$H_{0.1}$ There is no linear relationship between the strengths of structural and logical class dependencies.	$H_{1.1}$ There is a linear relationship between the strengths of structural and logical class dependencies.
Obj2	Is there a large overlap between semantic and logical coupling? Is there a linear relationship between semantic and logical class dependencies?	$H_{0.2}$ There is no linear relationship between the strengths of semantic and logical class dependencies.	$H_{1.2}$ There is a linear relationship between the strengths of semantic and logical class dependencies.
Obj3	Is there a linear relationship between structural and semantic class dependencies? Is there a statistically significant association between structural and semantic class dependencies?	$H_{0.3}$ There is no linear relationship between the strengths of semantic and structural class dependencies. $H_{0.4}$ There is no significant association between structural and semantic class dependencies.	$H_{1.3}$ There is a linear relationship between the strengths of semantic and structural class dependencies. $H_{1.4}$ There is a significant association between structural and semantic class dependencies.

Table 1.3: Summary of statistical tests per hypothesis

Hypotheses	Statistical test	What the test establishes
$H_{0.1}$, $H_{0.2}$, $H_{0.3}$	Spearman's rank correlation ρ in the **R** statistical environment (e.g., `cor.test(vector1, vector2, method = "spearman")`[4])	The strength of the link between two variables. In this case, the strength of the three types of static software dependencies (structural, logical and semantic) between OO software classes.
$H_{0.4}$	Chi-Squared and Fisher's exact independence test in the **R** statistical environment (e.g., `chisq.test(x)`, `fisher.test(x)`)	If there is a significant association between two variables. In this case pairs of types of class coupling in OO software.

C5 – Computational efficiency. Results have revealed that an identifier-based technique is more efficient than the corpora-based technique and reflects relatively analogous semantic similarity metrics.

C6 – Tool. This Thesis presents a tool developed and adopted to automate the extraction of semantic coupling metrics using a corpora based approach [6].

The following are those who have been anticipated to benefit from the contributions of this Thesis:

1. **Software testers**: Software testers are part of the beneficiaries of the work. n this Thesis as they will be able to focus their testing efforts assisted with knowledge about the interplay between software dependencies. As an example, when changes are made to one software class, other classes that have strong semantic or structural coupling to that class also need to be tested.

2. **Software comprehension, evolution and maintenance tool developers**:

   Software maintenance tools need to consider semantic and structural dependencies because these types of software relationships are essential in minimiz-

[6]The tool can be downloaded at: https://github.com/najienka/SemanticSimilarityJava

ing maintenance efforts, and propagate ripple effects of change. As such, their comprehension is important in improving software understanding and testing.

3. **Software maintainers**:

   Structural and semantic dependencies are propagators of ripple effects. "If developers are required to handle a large set of dependencies, they would miss a significant number" [34]. We have proposed techniques to bridge the gap between semantic and structural dependencies (software architecture).

1.7 Thesis structure

The following parts of this Thesis are structured as follows:

Chapter 2 presents an indepth review of the literature context of this study on several areas including change impact analysis, software maintenance, semantic, structural and logical coupling as well as the interplay between the three coupling types. Chapter 3 presents the research methodology of this study including the case study selection criteria, data extraction, coupling measurement and adopted statistical tests. Chapter 4 presents an empirical study carried out on 79 OSS projects to understand the interplay between structural and logical class dependencies in OO software.

Chapter 5 presents a study carried out on the same sample of 79 OSS projects to investigate the interplay between semantic and logical class dependencies. Chapter 6 probes the interplay between semantic and structural class dependencies in OO software and their overlap in the same sample of software projects.

Empirical studies in Chapters 4, 5 and 6 each include a worked example. The examples use software projects of varying sizes which are representative of the studied sample to illustrate the methodology adopted to resolve the research questions. Chapter 7 concludes this Thesis with directions for further research.

Section A.1 in Appendix A summarizes the software coupling metrics adopted in the literature till date. In Appendix B, Section B.1 shows the statistical outcomes of the empirical analysis carried out and described in Chapter 4. Section C.1 in Appendix C exhibits the outcomes of the empirical analysis carried out in the study presented in Chapter 5. In Appendix D, Section D.1 presents the statistical outcomes of the empirical study demonstrated in Chapter 6.

A list of cited research publications in this Thesis immediately follows Chapter 7. Afterwards, a glossary of frequently adopted software engineering terminologies in this Thesis is presented.

1.8 List of publications

Below is a list of the publications based on this Thesis:

1. N. Ajienka and A. Capiluppi. Semantic coupling between classes: Corpora or identifiers? In Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, page 40. ACM, 2016.

2. N. Ajienka and A. Capiluppi. Understanding the interplay between the logical and structural coupling of software classes. Journal of Systems and Software, Volume 134, December 2017, pages 120-137.

3. N. Ajienka, A. Capiluppi, S. Counsell. The interplay between semantic and logical coupling in object-oriented software. Empirical Software Engineering Journal, 2017, pages 1-35.

4. N. Ajienka, A. Capiluppi, S. Counsell, 2017. Managing Hidden Dependencies in OO Software: a Study Based on Open Source Projects. International Symposium on Empirical Software Engineering and Measurement, Toronto, Canada, 2017.

Chapter 2

State of the art

2.1 Introduction

In the previous chapter, we gave an introduction to the Thesis and outlined its structure. In this chapter, we describe related work to the research carried out in this Thesis. First related concepts are defined, such as empirical software engineering, software maintenance and evolution and OO software. Thereafter, issues in maintaining evolving software systems are outlined. Finally software measurement metrics, coupling metrics and types of coupling (interdependence) between OO software classes are described.

The chapter is structured as follows: in Section 2.2 we describe the issues around maintaining evolving software systems in order to ensure that they are usable, reliable and satisfy all necessary requirements. In Section 2.3 we explore the research in the areas of software change impact analysis and changeability prediction in OO software in relation to coupling. In Section 2.4 we discuss state of the art in coupling and introduce coupling sub-types in Subsections 2.4.1, 2.4.2 and 2.4.3. Finally, the three coupling types are synthesized and we explore studies on their interplay in Subsection 2.5. A summary of this chapter is presented in Section 2.6.

2.2 Software maintenance and evolution

Software systems frequently evolve to meet new change requirements [86] as almost all software that is useful and successful stimulates user-generated requests for change and improvements [27].

It is not uncommon for such systems to progress through years of development history, a number of developers, and a multitude of software artifacts including millions of lines of code. Software evolution is sometimes described as software maintenance [27]. However, the two terms refer to the same overall phenomenon but with different emphasis. The term “evolution” brings the focus on the gradual changes implemented into the system. When using the term “maintenance”, the emphasis is on maintaining stakeholder satisfaction [86] with the software over its application lifetime [47, 61].

Software maintenance tasks have been split into three categories:

- Corrective: correction of faults.
- Adaptive: improving usability in a changed environment.
- Perfective: improving performance, maintainability or other software attributes.

Burd and Munro [38] added a fourth category – Preventive: updating the software in order to improve upon its future maintainability without changing its current functionality. Von Mayrhauser and Vans [193] added two further categories – Reuse: understanding the problem, finding a solution in a closely related software component, locating and integrating that component in the system. Code Leverage: restructuring solution to fit into the use of predefined software components. This categorization has been reproduced in several software engineering (SE) research articles [44, 118, 119, 132, 158, 178].

In the context of OO software, Kung *et al.* [114] classified source code changes into four types:

- *Data change.* Updating the definition, declaration, access scope, access mode and initialization of any datum (a global variable, a local variable, or a class data member).

- *Method change.* A member function can be changed in various ways. This is classified into three types: component changes, interface changes, and control structure changes.

  Component changes include: (1) adding, deleting, or changing a predicate, (2) adding, deleting a local data variable, and (3) changing a sequential segment

  Structure changes include: (1) adding, deleting, or modifying a branch or a loop structure, and (2) adding, or deleting a sequential segment.

  The interface of a member function consists of its signature, access scope and mode, its interactions with other member functions (for example, a function call). Any change on the interface is called an interface change of a member function.

- *Class change.* Direct modifications of a class can be classified into three types: component changes, interface changes and relation changes. Any change on a defined/redefined member function or a defined data attribute is known as a *component change.* A change is said to be an interface change if it adds, or deletes a defined or redefined attribute, or changes its access mode or scope. A change is said to be a relation change if it adds, or deletes an inheritance, aggregation or association relationship between two classes.

- *Class library change.* These include: (1) changing the defined members of a class, (2) adding, or deleting a class and its relationships with other classes. (3) adding, or deleting a relationship between two existing classes, (4) adding, or deleting an independent class.

Software maintenance is the dominant lifecycle (Figure 2.1) activity for most practical systems [28]. The cost of software is essential. Improving software mainte-

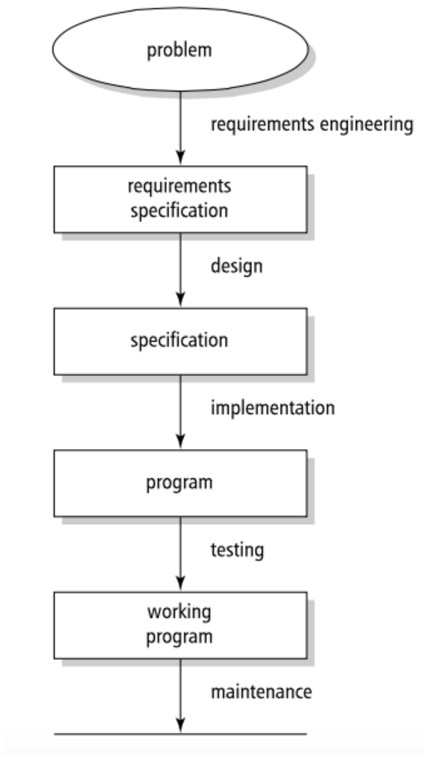


Figure 2.1: A simple view of software development ([189])

nance and development is an involved and costly task, with direct financial impact. In 2010 for example, global software expenditures amounted to $229 billion [29]. Maintenance, in its widest sense of "post deployment software support" is likely to continue to represent a very large fraction of total system costs [197]. As a result, it has become one of the most complex, crucial and costly disciplines within software engineering [106, 197]. Knowledge of the product(s) maintained, maintenance processes and communication skills is very important for achieving quality software and for improving maintenance and development processes [105].

Program comprehension is a major factor in providing effective software maintenance and enabling successful evolution of software [193]. Burd and Munro [38] proposed a number of metrics for the assessment of code maintainability while making the assumption that such assessments can at least partly be expressed in terms of comprehensibility.

The famous *eight* laws of software evolution by Lehman *et al.* have been widely used in software evolution research [136]. These laws emphasize the *E-type Systems*; software systems that solve a problem or implement applications in the real world [115]. Research suggests that several kinds of issues arise as these systems evolve. The 17 most important research challenges in software evolution have been outlined by Mens *et al.* [137].

2.3 Static change impact analysis (CIA)

Maintainability is largely determined by how well a system supports typical maintenance tasks, such as change impact analysis or regression testing [34]. Interestingly, prevalent OO programming languages have been reported to be a difficult case for impact analysis [197]. For instance, typical of OO systems is that implementations of methods or functions in classes tend to be much smaller than subroutines in a procedural system. Often, methods in one class just invoke methods in other classes, thus passing requests to another class.

A *ripple effect* is a phenomenon that affects other parts of a system on account of a proposed change. Small source code changes can have unexpected and non-local effects in software systems and this complicates software maintenance, e.g., adding a new method to a class may affect the behaviour of other methods throughout the program [170]. Hence, software developers or maintainers need to be aware of the methods or classes that will be affected by changes when a class or methods is changed [34, 181].

The task of an impact analysis technique is to estimate the (complete closure of) ripple effects and prevent side effects of a proposed change. The scope of the analyzed and estimated software artifacts may include requirements, design, source code, and/or test cases [102].

According to Ryder and Tip [170], "change impact analysis is a collection of techniques used to determine the effects of a set of source code changes"; this involves analyzing programs to detect software artifacts that are likely to be affected by a given change or a set of source code changes made in software systems. In other words "providing an ordering of software artifacts where ripple effects are more likely" [34] (i.e., *"if you change this, you will also have to change this/these".* Dependency analysis (impact analysis of software artifacts across the same levels of abstraction, e.g., source code to source code) and traceability analysis (impact analysis of software artifacts across different levels of abstractions, e.g., source code to documentation or design) are the two primary methodologies for performing impact analysis [103].

Dependency-based change impact analysis techniques can be divided into two (i.e., static and dynamic). Static techniques involve collecting structural, logical and semantic dependency information depending on whether the first feature or class to changed is known or not [74] given a change request. Static is preferred over dynamic techniques because dynamic analysis are associated with high costs, as it involves the monitoring and collection of dependency information of software during execution, it usually produces false negatives, i.e., some of the actual impact sets are not identified [181]. However, in cases where textually similar information cannot be identified in the classes, an estimated impact set based on semantic information will have low precision. In addition, in cases where there is no historical data or new artifacts have been introduced which are likely to be co-changed in following revisions, historical change information will yield an estimated impact set with a low precision[1]. This will also lead to unnoticeable dependencies [32].

A typical dependency based CIA technique takes a software entity in which a change is proposed or identified, and estimates other entities that are also potential change candidates, referred to as an estimated impact set, which is then compared with an actual impact set to evaluate the accuracy of the technique [102, 210]. It is noteworthy that this actual impact set can vary because changes can be imple-

[1]Precision is a measure of the exactness of a prediction set, while recall is a measure of its completeness [123].

mented differently [181].

Queille *et al.* [159] proposed an interactive process in which the programmer, guided by dependencies among program components (i.e., classes, functions), inspects components one-by-one and identifies the ones that are going to change.

Structural coupling measures have been used to support CIA in OO software [34, 198]. Wilkie and Kitchenham [198] investigated if classes with a high CBO (Coupling Between Objects) metric are more likely to be affected by ripple effects of change. The CBO for a class is the count of the number of other classes to which a class is structurally coupled. Although CBO was found to be an indicator of change-proneness, it was not sufficient to account for all possible co-changes.

Briand *et al.* [34] investigated the use of coupling measures and derived decision models for identifying classes likely to be changed during impact analysis in a commercial C++ system. The results of an empirical investigation of the structural coupling measures and their combinations showed that the coupling measures can be used to focus the underlying dependency analysis and reduce impact analysis effort because ripple effects are linked to structural coupling. On the other hand, the study revealed a substantial number of ripple effects, which are not accounted for by the highly structurally coupled classes.

Prior research [92, 162, 166] has proposed tools that can help navigate and prioritize system dependencies during various software maintenance tasks. For example, let us assume that during maintenance a given class A.java is identified to require modifications. Based on a static source code analysis, a tool could quickly provide the maintainer with a list of other classes to which class A.java is coupled to, ranked by their degree or strength of coupling with A.java. This would restrict the scope for a more refined, focused and detailed dependency analysis, and help to contain the explosion of dependencies that we usually observe in OO software [34].

Semantic coupling has also been applied during CIA. Poshyvanyk *et al.* [156] investigate the use of the conceptual coupling measures during change impact analysis. The paper reports the findings of a case study in the source code of the

Mozilla web browser. Kagdi *et al.* [102] combined conceptual and logical coupling for change impact analysis. Firstly, they select the first software entity (e1) for which IA needs to be performed. Secondly, they compute the conceptual coupling of e1 and other entities from the version of a software from where it is selected. After that, they mine commits (only commits relating to versions before the version in the previous step are considered) from the source code repository and compute the entities that are logically coupled to e1. Finally, they compute the estimated impact set (EIS) from the set of computed conceptual and logical couplings. They found out that combining both approaches out-performed using each approach on its own at the class and method level or granularity. Similarly, Sun *et al.* identified that combining semantic and logical class dependency information produces estimate impact sets with higher accuracy compared to individual techniques [74] and combinations (structural + logical, semantic + structural). However, the precision produced in their study for this combination was low (0.38) with recall of 0.61.

According to Oliva and Gerosa [148], an integrated view of both types of dependencies (logical and structural) should improve the effectiveness of both software change and maintenance activities. Research that aims to build visualization tools that comprise both types of dependencies will also be feasible. Table 2.1 summarizes the contributions and gaps identified in the studies evaluated in this section.

2.4 Software dependencies

Henderson-Sellers *et al.* [90] state that strong coupling complicates a system since a module is harder for a software engineer to understand, and change, if it is highly interrelated with other modules. “Software complexity can be reduced by designing systems with the weakest possible coupling between modules” [90].

Present-day software development projects result in systems containing a large number of interdependent (interrelated) programs or components [189]. Studies have demonstrated the relationship between coupling and external software quality

Table 2.1: Summary of findings on software dependencies and change impact analysis

Study	Contributions	Gaps
[159]	An interactive process in which the programmer, guided by dependencies among program components (i.e., classes, functions), inspects components one-by-one and identifies the ones that are going to change.	
[198]	Investigated if classes with high structural CBO (Coupling Between Objects) coupling metric values are more likely to be affected by change ripple effect.	CBO was found to be an indicator of change-proneness in general, but it was not sufficient to account for all possible co-changes [34].
[34]	Coupling measures can be used to focus the underlying dependency analysis and reduce impact analysis effort.	CBO highlighted the change-proneness of only a subset of classes [34].
[92, 162, 166]	Tools that can help navigate and prioritize system dependencies during various software maintenance tasks.	
[156]	Investigated the use of the conceptual coupling measures during change impact analysis.	It is possible that not all change dependencies can be captured by semantic coupling measurement only [1]. In Chapter 5 the interplay between logical and semantic dependencies is investigated.
[102]	Combined conceptual and logical coupling for change impact analysis.	CIA method based on evolution history is based on past operations and already existing change dependencies [175]. Therefore, it can lead to incorrect or incomplete results notably when new artifacts are introduced in the software.

attributes, such as fault-proneness, and the application of coupling to software maintenance tasks, such as impact analysis, finding bugs, guiding testing effort, and assessing change impact [163]. Coupling metrics are OO software metrics that measure the interdependence between a given component and other components in software projects.

According to Capiluppi *et al.* [41], coupling "is the union of all the includes, dependencies and functions calls (i.e., the common coupling) of all source files". Various coupling measures have been proposed over the years: *logical coupling* is a measure of the degree to which two or more components change together or co-evolve, based on the historical data of modifications; *semantic coupling* captures the degree to which the identifiers and comments from different classes relate to each other [21, 22, 73, 156, 160]; *structural coupling* is a measure of the structural or source code dependencies between software components e.g. classes, such as the number of method calls between them, or the inheritance relationships, while *dynamic coupling* is based on dynamic analysis of systems [11, 140, 163]. This Thesis is centered around three of these four types of coupling; structural, semantic and logical. In Subsections 2.4.1, 2.4.2 and 2.4.3 we discuss these three types of software coupling in the context of OO software.

2.4.1 Logical coupling

In software engineering, the term "logical coupling" (also known as *"change coupling"* [149]) captures the extent to which software artifacts have been co-changed and this information is derived by analyzing historical patterns, relationships and relevant information of source code changes mined from multiple versions (of software systems) in software repositories (e.g., Subversion and Bugzilla) [102, 203].

If two classes in an OO software system are frequently changed together during development, in other words, both classes have often been part of the same commits (changes simultaneously submitted to the version archive) this reveals an implicit dependency and since this information stems from the evolution of the software, this coupling concept is also called *evolutionary coupling*[2] [24, 25, 53, 150].

The concept of "logical coupling" was first introduced by Gall *et al.* [66] to detect implicit relationships between modules [53]. The technique that they proposed uses information from the CVS version control system to detect change dependencies between the modules of a software system. They used logical coupling to analyze the dependencies between the different modules of a large telecommunications software system and showed that the approach can be used to derive useful insights into the architecture of the system.

Kagdi [99] infers that by using the version history of a software system it is possible to identify or to predict software artifacts that are likely to co-change. This is done by analyzing revisions or commits, i.e., co-changed artifacts checked in together and associated metadata (e.g., date and text message), stored in software repositories (repositories store metadata such as user-IDs, timestamps, and commit comments in addition to the source code artifacts and their differences across versions). In this Thesis, we are only interested in commits that include files with the `.java` extension. We have studied software written in the Java programming language in this Thesis because Java is a popular language among open-source software projects [111, 138, 201] and because of the wide range of tools available to analyze software projects built using the Java programming language. The related terminologies associated with logical coupling are further defined.

A *"Software change history" (SCH)* is a set of change-sets (commits) submitted to the source-control repository during the evolution of a system in the time interval λ. A *revision* is a committed change in the history of a file or set of files. It is one *snapshot* in a constantly changing project. A *repository* is the master copy where source-control or version-control systems (e.g., CVS, Subversion, Github) store a projects full revision history. Each project has exactly one repository [6]. These terminologies are used in various chapters of this Thesis. Examples of repositories

[2]*Evolutionary* coupling and *Logical* coupling refer to the same concept (change dependency). For example, when file A changes, file B also changes.

are modern source-control systems, such as Github, Subversion and version-control systems such as CVS are used to infer logical couplings between artifacts [100]. "Logical couplings are then used to infer prediction rules of the form $e_1 \to e_2$. Such rules indicate that should the software entity e_1 change, the entity e_2 is also likely to co-change" [99].

Logical coupling is measured in terms of two established measures, *support and confidence* [24] introduced by Zimmermann *et al.* [208]. Support and confidence thresholds are used to filter logical dependencies (to filter out dependencies whose value of confidence is too low when extracting logical coupling data from version control systems) [150]. The support value of a coupling counts how often the two coupled software artifacts were changed together. Additionally, the confidence value of a coupling normalizes the support by the total number of changes of one of the artifacts and it is a measure of the strength of their co-evolution. While support is a symmetric metric, confidence is not because of its asymmetric normalization.

According to Yu [203], it should be noted that the value for support does not fully represent the co-evolution of two software components. Logical coupling is directional thus A $\to$ B and B $\to$ A will have different meanings. The former cause-effect rule states that changes made to file/class A resulted in changes in B, while the later states that changes in B caused changes in A. The support ratios for these two rules will be different.

The process of analyzing data in repositories to extract information on logical couplings of software artifacts is called mining software repositories (MSR) [102]. Several tools and approaches have been used in the literature to mine software repositories, logs, commits to extract logical couplings among software artifacts. In this Thesis we have adopted the CVSAnaly developed by other researchers [167, 168] tool to mine the repositories of the software projects studies to extract their change history. The tool is compatible with CVS, Subversion (SVN) and Git and has been widely used in MSR research. It extracts the data out of logs and stores them in SQL database tables.

Working at a finer granularity level, Zimmermann *et al.* [210] used CVS logs for detecting logical coupling between source code entities. Association rules based on itemset mining were then formed from the change-sets and used for change-prediction. Ying *et al.* [202] used a similar technique for identifying files that frequently change together.

Gall *et al.* [66] used window-based heuristics on revision logs for uncovering change patterns, and [72] for studying characteristics of different types of changes. Sun *et al.* [182] classified various change types (CT) at different granularity levels of OO programs (class, method and field) - particularly for Java programs. Tables 2.2, 2.3 and 2.4 show these change types (CT) and their meanings. Table 2.2 demonstrates the types of modifications that can be made to classes in OO software. Table 2.3 demonstrates the types of changes that can be made to methods or functions in software. Lastly, Table 2.4 shows the types of changes that are made to fields in software.

Table 2.2: Class change types

CT	Meanings
AC	Add a common class (a class declaration)
DC	Delete a class (including all its members)
IAC	Increase "accessibility" of the class("private" is modified to "public")
DAC	Decrease "accessibility" of the class("public" is modified to "private")
AFC	Add modifier "final" to the class
DFC	Delete modifier "final" of the class
ASC	Add modifier "static" to the class
DSC	Delete modifier "static" of the class
AAbC	Add modifier "abstract" to the class
DAbC	Delete modifier "abstract" of the class
CNC	Change name of the class
APC	Add a "parent" to the class
DPC	Delete a "parent" of the class
CPC	Change the "parent" of the class

2.4.2 Structural coupling

According to Poshyvanyk and Marcus [155], structural coupling is measured or determined by program analysis and it is directional [203]. There is a *directed*

Table 2.3: Method change types

CT	Meanings
AM	Add a common method (a method declaration)
DM	Delete a method (including all its statements)
IAM	Increase the "accessibility" of the method("private" is modified to "public")
DAM	Decrease the "accessibility: of the method("public" is modified to "private")
CM	Change the statements within the method body
AFM	Add modifier "final" to the method
DFM	Delete modifier "final" of the method
ASM	Add modifier "static" to the method
DSM	Delete modifier "static" of the method
AAbM	Add modifier "abstract" to the method
DAbM	Delete modifier "abstract" of the method
CRM	Change "return type" of the method
CNPM	Change "name of the parameters" of the method
CPM	Change "parameters" of the method except for the CNPM
CNM	Change "name" of the method

Table 2.4: Field change types

CT	Meanings
AF	Add a common field (a field declaration)
DF	Delete a field
IAF	Increase "accessibility" of the field("private" is modified to "public")
DAF	Decrease "accessibility" of the field("public" is modified to "private")
AFF	Add modifier "final" to the field
DFF	Delete modifier "final" of the field
ASF	Add modifier "static" to the field
DSF	Delete modifier "static" of this field
CTF	Change "type" of the field
CNF	Change "name" of the field

dependency between two classes A and B if A depends on B in such a way that A is not operational without component B [71].

Singh *et al.* have stated that “an incoming coupling exists when a class provides its services to other classes”. “While an outgoing coupling exists when a class uses the services of another class” [177]. They represent incoming coupling with an arrow from the caller class to the called class and results in their study show a low negative correlation between incoming coupling and change as well as outgoing coupling and change. In their example representation C⇒A⇒B; class A is incoming coupled with C because its method provides service to a method in C. On the other hand, A is outgoing coupled with class B because its method used a method of class B.

In the case of the Java programming language; this means that A would not compile (converted into a machine-code or lower-level form in which the program can be understood and executed by a computer) in the absence of B. Furthermore, the relationships “class A depends on class B” and “class B depends on class A” have different effects on software evolution. Geipel and Schweitzer [71] infer that if A structurally depends on B, changes made to B can lead to changes to A, but not the other way round. However, this will not be the case if the coupling type is symmetric or bi-directional (e.g., semantic coupling, described in Section 2.4.3).

Geipel and Schweitzer [70] model OO software as a network of classes which are dependent on each other. Consequently, this network may be represented as a call graph containing nodes and edges or as an adjacency matrix. They refer to the dependency matrix as D were $D_{i,j} = 1$ means that i depends on j. $D_{i,j} = 0$ on the other hand is interpreted as independence. There is a dependency between classes i and j if i extends or implements j, i calls a method provided by j, i references members of j, or i uses j as member or variable [70]. In each of these cases $D_{i,j}$ is set to 1. It is important to note that D is an asymmetric matrix and D is only valid at one particular point in time because software systems evolve, so does their structure or dependency network: classes are changed, old ones removed, new ones added to the system and so too dependencies between classes.

The constructs of most OO programming languages such as C, C++, and Java can induce various types of structural relationships [63, 157, 197]. A method calls another method, a class extends another class, or a class aggregates objects of another class - all of these call relationships create a direct structural dependency between two classes. These static structural code dependencies are most frequently used when analyzing or leveraging structural coupling [24]. In this study, we have leveraged these metrics when computing the strength of the structural coupling between two classes.

Structural coupling (simply called "coupling" in some studies [117, 146, 203, 204] is still considered to be an imprecise measure of software complexity [146]. Many researchers have empirically investigated and identified the relationship between structural coupling and the external quality factors of software systems such as fault-proneness and maintenance [21, 84], change impact analysis [34, 74, 102, 156, 156, 163], re-engineering, reuse, change propagation, and clone management [21].

The well known CBO metric (coupling between objects) has been proposed in the past to quantitatively define the structural coupling of a class by Chidamber and Kemerer [46]. Wilkie and Kitchenham [198] state that the "CBO for a class is a count of the number of other classes to which it is (structurally) linked". In these studies ([13, 122, 126]) they performed change predictions implicitly using static (structural) coupling metrics initially proposed by [46], such as coupling between objects (CBO), response for a class (RFC), lack of cohesion (LCOM), weighted methods per class (WMC) and in addition, sources lines of code (SLOC) as the input for their prediction models.

Briand *et al.* [34] analyzed the C++ classes in a commercial OO software. Using its OO software metrics, they evaluated the impact of future changes. The set of OO coupling measures explored in the study [34] include the: CBO (coupling between objects); MPC (Message passing coupling, i.e., number of references between classes); the number of static invocations in class C of methods in class D); DAC (Data abstraction coupling, i.e., the number of attributes in a class C that have class D as their type); and PIM (number of method invocations in C of methods in D. . Takes polymorphism into account).

Briand *et al.* [32] developed a framework to compare various forms of coupling. They stated that it is often difficult to determine how various coupling measures relate to one another and for which application they can be used. As a consequence, it is very difficult for practitioners and researchers to obtain a clear picture of the state of the art in order to select or define measures for OO systems [32, 33]. Morasca *et al.* [143] also state that "a large number of measures have been proposed in the literature to measure software attributes (e.g., size, complexity cohesion and coupling)". Those proposed measures might not be adequate for the software attributes they purport to measure. The authors showed how a hierarchical axiomatic framework can be constructed to support the definition of *consistent* measures for a given software attribute at different levels of measurement [143].

2.4.3 Semantic coupling

Some studies [16, 21, 22, 23, 103, 156, 160] have used the term *"semantic"*, while others have used the term *"conceptual"* [73] to describe the same concept. Poshyvanyk *et al.* [156] state that *conceptual coupling* captures the degree to which the identifiers and comments from different classes relate to each other. Gethers *et al.* [73] add a twist to the definition and state that conceptual coupling captures the extent to which domain concepts and software artifacts are related to each other. However, both definitions have things in common. They are limited to the underlying meanings of unstructured text in the source code of software entities (e.g., classes) and how these underlying meanings relate to each other. Furthermore, this relationship is derived in the form of metrics (-1 to 1, where 1 = high semantic coupling [155]).

Identifiers used by programmers for names of classes, methods, or attributes in source code or other artifacts contain important information and account for approximately half of the source code in software [102]. These names often serve as a starting point in many program comprehension tasks. Hence, it is essential that these names clearly reflect the concepts that they are supposed to represent, as self-documenting identifiers decrease the time and effort needed to acquire a basic comprehension level for a programming task [102].

In most studies, semantic coupling was used (alongside structural or logical coupling) to support a number of software activities such as change impact analysis [102, 103], file coupling [75], feature coupling [73, 163], fault proneness [7] and modularisation [17, 18, 22].

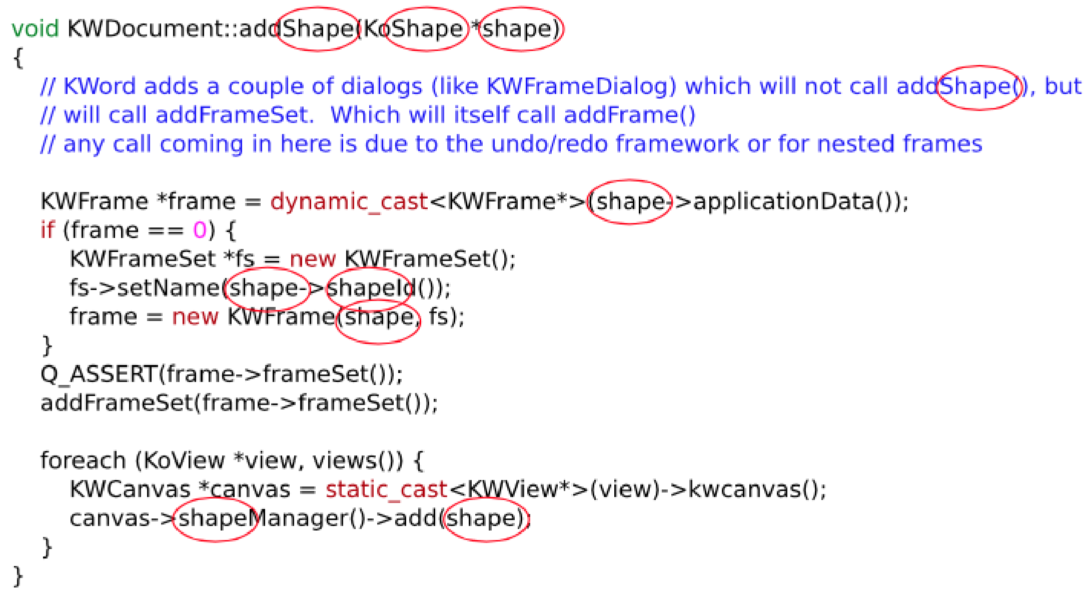
Kuhn *et al.* [113] proposed the use of information retrieval techniques to exploit linguistic information found in source code, such as identifier (i.e., class or method) names and comments. They introduce semantic clustering, a technique based on latent semantic indexing (LSI) and clustering to group software artifacts that use similar vocabulary. A highlight of their approach is that it is language independent as it works at the level of identifier names.

The underlying mechanism widely used in majority of the studies to extract and analyze the semantic information from the source code is Latent Semantic Indexing (LSI)[3] or Vector Space Model (VSM)[4]. The steps taken to extract and analyze the semantic information from source code as explained in [163] and adapted by other studies - are explained further in Chapter 3.

An example of semantic coupling between methods is presented in [102]. The study shows how two methods addShape() and removeShape() shown in Figures 2.2 and 2.3 are highly semantically related (semantic similarity = 0.78) due to them containing similar terms (the term "shape" is highlighted to visualize its contribution to the computed metric). A tool to automate the semantic coupling metrics extraction process for Java classes in OO software systems has not been developed as at the time of writing this Thesis and compared to previous studies, we have addressed this gap by developing a tool. Rilling *et al.* [165] identified diverse tasks in

[3]LSI is an advanced information retrieval (IR) technique which identifies the relationships between terms and concepts in unstructured text.

[4]Transforming documents into vectors and computing their cosine similarities



```
void KWDocument::addShape(KoShape *shape)
{
   // KWord adds a couple of dialogs (like KWFrameDialog) which will not call addShape(), but
   // will call addFrameSet.  Which will itself call addFrame()
   // any call coming in here is due to the undo/redo framework or for nested frames

   KWFrame *frame = dynamic_cast<KWFrame*>(shape->applicationData());
   if (frame == 0) {
      KWFrameSet *fs = new KWFrameSet();
      fs->setName(shape->shapeId());
      frame = new KWFrame(shape, fs);
   }
   Q_ASSERT(frame->frameSet());
   addFrameSet(frame->frameSet());

   foreach (KoView *view, views()) {
      KWCanvas *canvas = static_cast<KWView*>(view)->kwcanvas();
      canvas->shapeManager()->add(shape);
   }
}
```

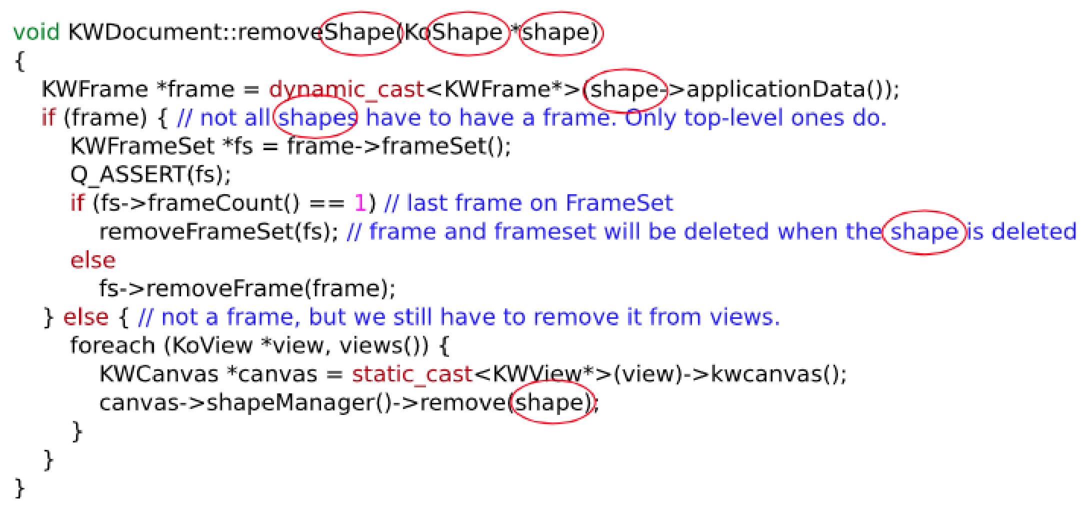
Figure 2.2: A method named addShape() from KOffice showing the conceptual information that is latent in (some of the) identifier names, adapted from [102]

software maintenance where the use of semantic technologies can be beneficial, such as traceability, system comprehension, software artifact analysis, and information integration.

2.5 Interplay on the coupling axes

OO software usually evolves to satisfy modification requirements [86]. Understanding the characteristics and initiators of changes aids testers and system designers to improve the quality of software [194]. In Sections 2.4.1, 2.4.2 and 2.4.3 the three software coupling dimensions explored in this Thesis are introduced and described.

In this section, we will evaluate prior research on the interplay between the coupling dimensions along three different axes; LSt, LSe and StSe described in Section 1.2 of Chapter 1.



```
void KWDocument::removeShape(KoShape *shape)
{
   KWFrame *frame = dynamic_cast<KWFrame*>(shape->applicationData());
   if (frame) { // not all shapes have to have a frame. Only top-level ones do.
      KWFrameSet *fs = frame->frameSet();
      Q_ASSERT(fs);
      if (fs->frameCount() == 1) // last frame on FrameSet
         removeFrameSet(fs); // frame and frameset will be deleted when the shape is deleted
      else
         fs->removeFrame(frame);
   } else { // not a frame, but we still have to remove it from views.
      foreach (KoView *view, views()) {
         KWCanvas *canvas = static_cast<KWView*>(view)->kwcanvas();
         canvas->shapeManager()->remove(shape);
      }
   }
}
```

Figure 2.3: A method named removeShape() from KOffice showing the conceptual information that is latent in (some of the) identifier names, adapted from [102]

2.5.1 Interplay on the logical and structural (LSt) coupling axis

In a seminal study on the Linux kernel, 12 classes (written in C) were studied from the kernel. It was shown that structural coupling between classes causes them to be co-changed and that plays an important role in the measurement of co-evolution (i.e., coupling leads to co-evolution) [203].

D'Ambros *et al.* [55] studied the relationship between co-changes and bugs, and found a relationship. The authors enriched their findings by using OO software metrics, but the correlation between these metrics and co-change patterns were not analyzed. In this Thesis, we analyze the linear correlation between the degree of the structural and logical as well as semantic coupling between class pairs in a sample of OO software projects.

Kafura *et al.* [91, 98] found strong correlations between software metrics based on information flow among system components and the number of changed source lines in the Unix operating system. Kitchenham *et al.* [110] were not able to validate the findings in [98]. However, they found that change was related to other metrics such as fan-out, size and number of branches.

Using a syntactic complexity metric (SynC), Basili *et al.* [15] observed a correlation between this metric and software change in 19 student developed software projects. Binkley and Schach [31] found out that their coupling dependency metric (CDM) and change were correlated. In summary, as discussed by Shepperd and Ince [176], some researchers found strong relationships between static measures and change or defects but other researchers observed weaker relationships. Most of those studies mentioned by Shepperd and Ince involved correlation analysis or linear regression models.

Gall *et al.* [66] were the first to use co-evolution to represent coupling. They developed a technique called CAESAR for detecting change patterns and applied it to a large Telecommunication Switching System with a 20-version history. Their approach identified logical dependencies among modules (hidden in source code) in such a way that potential structural shortcomings could be identified and further examined, pointing to restructuring or re-engineering opportunities. Rather than dealing with millions of lines of code they used structural information about programs, modules, and subsystems, together with their version numbers and change reports for a release to discover common change behavior (i.e., change patterns) of modules and identified potential dependencies among modules, and validated these potential dependencies by examining change reports that contain specific change information for a release.

Zimmermann *et al.* [208] analysed the revision history of individual components and functions to detect the fine-grained coupling (they noticed that components with strong co-evolution also have strong structural coupling but did not provide empirical evidence). In this study, we empirically investigate the interplay between structural, logical and semantic coupling using statistical methods described in Chapter 3.

Differently from this study which focuses on the class level of granularity in OO software, at the method or feature level of granularity Mondal *et al.* [142] have investigated the effects of method sharing (among different functionality) on method co-changeability and source code modifications. They proposed and empirically evaluated two software metrics, (i) COMS (Co-changeability of Methods), and (ii) CCMS (Connectivity of Co-changed Method Groups) and investigated the impact of CCMS on COMS and source code modifications. Their comprehensive study on hundreds of revisions of six open-source software projects covering three programming languages (Java, C and C#) suggests that higher CCMS causes higher COMS as well as increased source code modifications.

Revelle *et al.* [163] argued that there is a growing focus on the study of features in software, and features (methods) are often implemented across multiple classes, meaning class-level coupling measures are not applicable. They asked the question "is measuring coupling at the feature-level also useful?" and defined new feature coupling metrics based on structural and textual source code information and extend the unified framework for coupling measurement to include these new metrics. The metrics proved to be good predictors of fault-proneness and useful tools for developers performing feature-level software maintenance tasks.

Hanakawa [87] proposed a visualisation technique and software complexity metrics for software based on co-evolution and coupling. The idea was that *modules including strong coupling should have strong co-evolution, thus co-evolve or co-change.* Yu [203] conducted a study on 12 Linux kernel modules, comparing 12 pairs of co-evolution data and structural coupling data and based on findings – established that a linear relationship exists between co-evolution and structural coupling and thus proved that the dependencies between software components induced via the system architecture have noticeable effects on component co-evolution. However, as highlighted in Chapter 1, the p-value used to evaluate the significance of the linear correlation results was 0.1 (10% error margin) and the author acknowledged this as one of the threats to the internal validity of the study. Evaluating the linear relationship (correlation) between logical and structural coupling strengths on a different sample of software projects with a p-value of 0.05 or 0.01 could produce distinct findings [192]. In this Thesis we attempt to resolve this issue with a lower p-value as described in Chapter 3.

Recent studies [64] [71] [147] [148] have shown that it is possible that structural and logical coupling are caused by other subtle and implicit types of relationships (e.g., semantic dependencies) as they have identified a significant number of logically coupled classes without structural coupling links between them and *vice versa.* Fluri *et al.* [64] investigated the degree to which co-changes are caused by structural changes (source code changes) and textual modifications (e.g., software license updates and white-spaces between methods spaces). A preliminary evaluation involving the compare plugin of Eclipse showed that more than 30% of all change transactions did not include any structural change. Therefore, more than 30% of all change transactions have nothing to do with structural coupling. They also found that more than 50% of change transactions had at least one non-structural change. They hypothesize that this could be the result of code ownership/commit habit (a developer works all day in his files and commits everything by the end of the day) and frequent license changes.

Oliva and Gerosa [148] analyze Java files of the first 150 thousand commits from Apache software repository to investigate and quantify the proportion of logical dependencies that involve non-structurally related elements and the proportion of structural dependencies that involve non-logically related elements. They concluded that in 91% of the cases logical dependencies involve non-structurally related files, most logical dependencies are not directly caused by structural dependencies. Furthermore, structural dependencies very frequently involve files that are not logically related, hence there is a very small intersection between sets of structural and logical dependencies.

Geipel and Schweitzer [71] analyze the link between structural dependency and co-change. Their study takes into consideration *only* the latest code snapshot when extracting structural dependencies. They argue that structural dependencies between two classes i and j are somewhat "stable" from the creation of the younger class until the removal of either i or j. This assumption did not hold for the projects studied in [147]. According to their results, many structural dependencies are never involved in change propagation and state that if most active 10% of the dependencies are responsible for over 70% of the co-changes, as is the case in Eclipse, then the co-change behaviour is hardly a mirror image of the dependency structure.

Building on their previous work [148] and other studies [71] [89] [127], Oliva and Gerosa [147] conducted a study in which they investigate the influence of structural dependencies on change propagation in four Java open-source software of different sizes in terms of number of classes. They consider that there is a structural dependency from f1 to f2 only if f1 depends on “something” that changed in f2 (e.g., a field definition or a method’s body), and their results indicate that the rate with which an artifact co-changes with another is higher when the former structurally depends on the latter. Their results indicate that there is a causal relationship between structural coupling and co-evolution (co-evolution is dependent on structural coupling [203]).

2.5.2 Interplay on the logical and semantic (LSe) coupling axis

It is important to note that overall the measurement of semantic coupling is more affected by the difference in granularity than logical coupling. For the conceptual couplings, going from the coarse granularity of files to the finer granularity of methods resulted in the reduction of the sizes of the documents. The documents are reduced in terms (and frequency). That is, a corpus for a class is typically much “bigger” than a corpus for a method [102]. For logical coupling some commits did not contain changes made to methods while some did not contain changes made to classes, so there is no way to map changes made to classes and methods. This informs our choice of studying the software artifacts at the class level of granularity in this Thesis.

Kagdi *et al.* [102] in their study on combining logical and semantic coupling to support change impact analysis found that finer granularity decreases accuracy of all approaches, it does not prevent the combination of the two from out-performing the standalone techniques. That is, the gain acquired by combining conceptual and logical coupling exists regardless of the granularity (file-level and method-level) considered in the study.

2.5.3 Interplay on the structural and semantic (StSe) coupling axis

To improve maintainability, software systems are usually organized into subsystems using the constructs of packages or modules. However, during software evolution the structure of the system undergoes continuous modifications, drifting away from its original design, often reducing its quality. Bavota *et al.* [17] proposed an approach for helping maintainers to improve the quality of software modularization. The proposed approach analyzes the (structural and semantic) relationships between classes in a package identifying chains of strongly related classes. The identified chains are used to define new packages with higher cohesion than the original package.

It can be inferred from the studies highlighted in Sections 2.5.1, 2.5.2 and 2.5.3 that static metrics are generally associated with change-proneness. An advantage of studying coupling measures in relation to co-changes is that they are inherently related to ripple effects since common changes are usually due to relationships between classes [34, 174]. However, Geipel and Schweitzer [71] state: "the question about the causes of change propagation has been skipped by many researchers [89, 126, 185, 202, 210] in favour of a predictive approach in which these causes are implicitly contained in a prediction function or as inputs to a machine learning algorithm".

These studies on the relationship and interplay between structural and logical coupling in Section 2.5.1 have shown that the sets of structurally coupled pairs of classes are always smaller than the sets of co-changed pairs. According to Geipel and Schweitzer [71], this indirectly means that any model that tries to infer structural coupling from logical coupling or co-evolution will produce a lot of false positives. On the other hand, using the structural coupling information between pairs of classes to infer their future co-change is a more realistic objective [147]. The hazardous effects of change propagation to software maintenance have been discussed in seminal works [36, 131, 152]. If software is difficult to maintain or modify to keep up with user requirements, this will add to its swift deterioration. A process Parnas [152] refers to as *"software aging"*.

Tables 2.5 (logical and structural), 2.6 (logical and semantic) and 2.7 (structural and semantic) summarize the contributions and gaps identified along the interplay axes evaluated in this section.

Table 2.5: Summary of findings on the interplay between logical and structural coupling

Study	Contributions	Gaps
[203]	Structural coupling between components leads to their logical coupling.	Studied only files written in C and in one project. The p-value adopted to evaluate the significance of the results was 0.1 (a 10% error margin) [203] and the author acknowledged this a one of the threats to validity of the study. Replicating the study on a larger sample of projects and evaluating the correlations results with a p-value of 0.05 or 0.01 can produce distinct findings. This gap is resolved in Chapter 4.
[55]	Found a relationship between co-changes and bugs.	The correlation between these metrics and co-change patterns were not analyzed.
[91, 98]	Strong correlations between software metrics based on information flow among system components and the number of changed source lines in the Unix operating system.	Results from a different study [110] did not validate the information flow metrics used [98].
[110]	Change is related to metrics such as fan-out, size and number of branches.	

Continued on next page

Table 2.5 – *Continued from previous page*

Study	Contributions	Gaps
[15]	Using a syntactic complexity metric (SynC), they observed a correlation between this metric and change in 19 student software projects.	
[31]	Proposed coupling metric (CDM) and change are correlated.	
[66]	Identified logical dependencies among modules (hidden in source code) in such a way that potential structural shortcomings can be identified and further examined, pointing to restructuring or re-engineering opportunities	
[208]	Components with strong co-evolution also have strong structural coupling.	No empirical evidence provided. This gap is investigated in Chapter 4.
[64]	More than 30% of all change transactions did not include any structural change and more than 50% of change transactions had at least one non-structural change. They hypothesize that this could be the result of code ownership/commit habit (a developer works all day on their files and commits everything by the end of the day) and frequent license changes.	No study on semantic coupling and co-change of classes. This gap is addressed in Chapter 5.

Continued on next page

Table 2.5 – *Continued from previous page*

Study	Contributions	Gaps
[71]	Argue that structural dependencies between two classes i and j are somewhat “stable” from the creation of the younger class until the removal of either i or j. This assumption did not hold for the projects studied in [147].	The question about the causes of change propagation has been skipped by many researches in favour of a predictive approach in which these causes are implicitly contained in a prediction function or as inputs to a machine learning algorithm [89, 126, 185, 202, 210].
[147]	There is a link between coupling and co-evolution and that co-evolution is dependent on coupling.	Only four OSS in studied sample of software projects (the results lack the power of generalizeability). In this Thesis we have studies a larger sample of 79 OSS projects as described in Chapter 3.
[71, 147]	Any model that tries to infer structural coupling from logical coupling or co-evolution will produce a lot of false positives. On the other hand, using the structural coupling information between pairs of classes to infer their future co-change is a more realistic objective.	Structural dependency measurement alone cannot predict co-change with a high precision as structural dependencies do not capture semantic or hidden dependencies [1, 32]. In Chapter 6, we investigate the interplay between structural and semantic coupling. This will contribute to the understanding of the degree to which structural and semantic class dependencies overlap in OO software.

Table 2.6: Summary of findings on the interplay logical and semantic coupling

Study	Contributions	Gaps
[64]	More than 30% of all change transactions did not include any structural change and more than 50% of change transactions had at least one non-structural change. They hypothesize that this could be the result of code ownership/commit habit (a developer works all day on their files and commits everything by the end of the day) and frequent license changes.	No study on semantic coupling and co-change of classes. This gap is addressed in Chapter 5.
[102]	Finer granularity decreases accuracy of conceptual and logical coupling measurement approaches. For logical coupling some commits did not contain changes made to methods while some did not contain changes made to classes, so there is no way to map changes made to classes and methods. for conceptual coupling, the corpus (term documents) generated at the method level are smaller.	No study on the interplay between semantic and structural coupling. These types of coupling propagate ripple effects. Also, the authors did not investigate why an analysis of either semantic or structural coupling will not yield a complete or almost precise estimate of change impact sets. This gap is addressed in Chapter 6.
[174]	An advantage of studying coupling measures in relation to co-changes is that they are inherently related to ripple effects since common changes are usually due to relationships between classes	No study on the interplay between semantic coupling and logical coupling. This gap is addressed in Chapter 5.

Continued on next page

Table 2.6 – *Continued from previous page*

Study	Contributions	Gaps
[71, 147, 148]	It is possible that coupling and co-evolution are caused by other types of relationships (e.g., conceptual dependencies)	(1) Their computation of structural coupling metrics between classes was not based on actual facts but based on estimates. Therefore reliability of findings is questionable [148]. Our computation of structural coupling takes a different methodology described in Chapter 3. (2) The interplay between semantic and logical dependencies has not been studied [71, 147, 148]. This gap is addressed in Chapter 5. (3) There is a need for a tool to automate the measurement of semantic coupling between corpora of software classes. Such a tool will ease the process for non-experts in data mining and information retrieval (IR) techniques, provide a standard unified framework for the extraction process as well as promote better comparison of results in this field [165]. We address this gap with the development and evaluation of a tool in Chapter 5.

Table 2.7: Summary of findings on the interplay between structural and semantic coupling

Study	Contributions	Gaps
[71, 147]	Any model that tries to infer structural coupling from logical coupling or co-evolution will produce a lot of false positives. On the other hand, using the structural coupling information between pairs of classes to infer their future co-change is a more realistic objective.	Structural dependency measurement alone cannot predict co-change with a high precision as structural dependencies do not capture semantic or hidden dependencies [1, 32]. In Chapter 6, we investigate the interplay between structural and semantic coupling. This will contribute to the understanding of the degree to which structural and semantic class dependencies overlap in OO software.
[163]	A combination of structural and textual source code information for fault-proneness prediction at the feature-level.	Prior research has demonstrated the inefficiencies related to measuring semantic coupling at the feature or method level of granularity. In this study we have measured semantic coupling at the class level of granularity.

2.6 Summary of the chapter

In this chapter we have reviewed and summarized the state of the art in the areas of software maintenance and dependencies and identified various gaps as listed in Tables 2.1, 2.5, 2.6, and 2.7. The studies on change impact analysis have probed ripple effects to infer a list of possible components that will need to be changed using coupling metrics and CIA techniques; however they have not focused on the starting point from which these changes propagate; the direct dependencies between class pairs.

There have also been studies on the interplay between structural and logical coupling but most of these studies are done on an average of 10 open-source systems, thus there is a need to make these results more generalizeable by conducting large scale empirical studies on more systems. While studies have been carried out on the interplay between structural and logical dependencies, none have been carried out on the interplay between semantic and (1) structural or (2) logical dependencies as rightly identified by Oliva and Gerosa [147, 148]. They state that “investigating the interplay between logical, structural dependencies and other kinds of dependency (e.g. conceptual [155]) should be fertile research topics”. This also brings about the need for a tool to identify and extract the semantic coupling metrics for pairs of software classes automatically as well as comparing this tool to other simple Information Retrieval (IR) techniques. The benefits of the application of semantic technologies to software maintenance have been highlighted in [165].

It is important that software co-changes should be predictable with a high precision to curtail maintenance efforts and costs. “To limit change propagation, we should make them as foreseeable as possible” [147]. In Chapter 3, the methodology of this study is described in detail.

Chapter 3

Methodology

3.1 Introduction

In previous chapters, this Thesis introduced three static software dependency types (i.e., structural, logical and semantic) including existing gaps identified in prior research in the area of the interplay between these dependency types. Studying their interplay requires adequate data sources as well as validated measurement tools that can be adopted to automatically measure the three dependency types. This chapter will explain in detail the selection of the analyzed case studies (OO software projects); data extraction and tools, measurement techniques adopted to capture the three types of dependencies between classes in the case studies and finally the empirical tests conducted to resolve the research questions described in Chapter 1. At the end of this chapter will be a summary followed by an introduction of the empirical studies conducted along the three coupling axes described in Section 1.3 of Chapter 1.

3.2 Data sampling

Open-source software (OSS) is software provided with a licence which permits anyone to study, modify and distribute the software to anyone for any purpose. In this study, we have analyzed OO OSS, using their archived source code. This approach will provide insights into how developers and maintainers maintain software coupling and in turn provide knowledge into coupling trends across a variety of software projects of different domains. We use tools described in Section 3.4 of the chapter, to extract the structural, logical and semantic dependency metrics.

Prior to identifying sources of open-source software data, we defined a set of selection criteria for our study sample, as follows:

1. OSS projects which (i) provide public access to source code and (ii) use a version control system that allows us to extract the historical information [52];
2. Were implemented in Java to allow the extraction of the structural coupling between classes, since structural coupling varies between languages [148];
3. Randomly sampled projects;
4. Multiple revisions/commits ($>$ 20 revisions in order to exclude trivial projects)[1], and a relatively long history log;
5. Have a large group of users.

Leveraging the FlossMole project [57, 95, 173], we used its latest available data dump to determine the population of GoogleCode: a total of 2,593,222 projects were listed in the database[2]. Given their language descriptions, we extracted the subset of Java projects from that population, obtaining 49,459 Java projects. Each project in the subset was given a unique ID: the sizing of the sample was achieved by considering a 95% confidence level ($\alpha = 0.05$) resulting in a random sample of 380 IDs extracted, and linked to the Java projects' names.

[1]Prior research [107] shows that 75% of OSS projects on Github have over 20 commits and 90% have less than 50 commits. We selected projects with above 20 commits to include a mix of projects with varying levels of activity in our sample, improve generalizeabiliy of the study as well as extract substantial change history to understand logical coupling.

[2]The data is available at http://flossdata.syr.edu/data/gc/2012/2012-Nov/

According to Geipel and Schweitzer [71] "Java, unlike other popular languages such as C++ was designed from scratch to be an OO language. Each class is defined in a separate file. For this reason, file changes can be directly mapped to class changes. Furthermore, Java enjoys a high popularity in the Open Source community: On SourceForge (http://www.sf.net) — the largest Open Source incubator site — Java is used in approximately 25 percent of the projects (as at August 2007). This makes Java the most popular language used [37, 138] with a high availability of tools for analyzing Java source code.

Below is the formula we used in calculating the sample size of the OSS forge:

$$Samplesize = \frac{N}{1 + (N(e^2))} \tag{3.1}$$

Where:
e = error margin (0.05)
N = total population

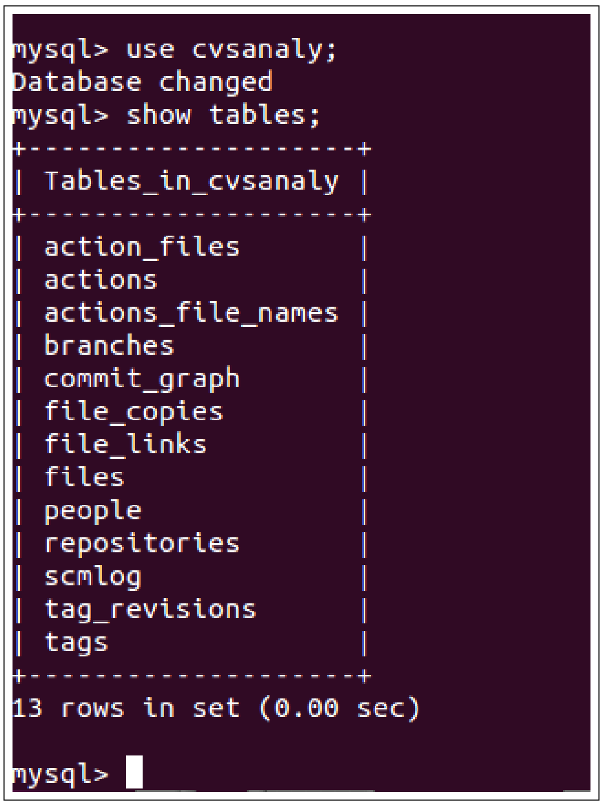
3.3 Data extraction

The CVSAnalY tool[3]. adopted in this study helps a researcher to manually inspect version control systems (e.g., CVS) meta-data from OSS projects repositories [188]. It automatically extracts and stores this data in a MySQL database. It currently supports the CVS, SVN and Git version control systems. The data it extracts includes developers' actions during software corrective maintenance activities [168].

Figure 3.1 shows the names of tables automatically generated by CVSAnalY in a MySQL database while Figure 3.2 illustrates the generated database schema. The database schema "is the artifact responsible for maintaining the integrity of stored data" [133] in the database. As an example, one of the tables is the *actions*

[3]The CVSAnaly tool has been developed by other researchers and adopted in this study when measuring logical coupling

table described in Figure 3.3. This table accommodates information regarding the changes made to artifacts including the file identifier, commit identifier and branch identifier all from a version control system. The *repositories* table contains the IDs and names of repositories mined.



```
mysql> use cvsanaly;
Database changed
mysql> show tables;
+--------------------+
| Tables_in_cvsanaly |
+--------------------+
| action_files       |
| actions            |
| actions_file_names |
| branches           |
| commit_graph       |
| file_copies        |
| file_links         |
| files              |
| people             |
| repositories       |
| scmlog             |
| tag_revisions      |
| tags               |
+--------------------+
13 rows in set (0.00 sec)

mysql> 
```

Table 3.1: CVSAnalY database tables

The first phase of the data extraction activity was focused on obtaining the meta-data (e.g, name of developers, date and time of changes to the classes, etc.) of each OO software project in the sample. The repository of each project was downloaded and stored, with its meta-data, using the CVSAnalY[4] tool[5]. The meta-data exposed the list of revisions for each class, and for each project as a whole.

[4]http://metricsgrimoire.github.io/CVSAnalY/

[5]Installation steps can be found at: https://sites.google.com/site/arnamoyswebsite/Welcome/updates-news/howtoinstallandruncvsanaly2inubuntu1110

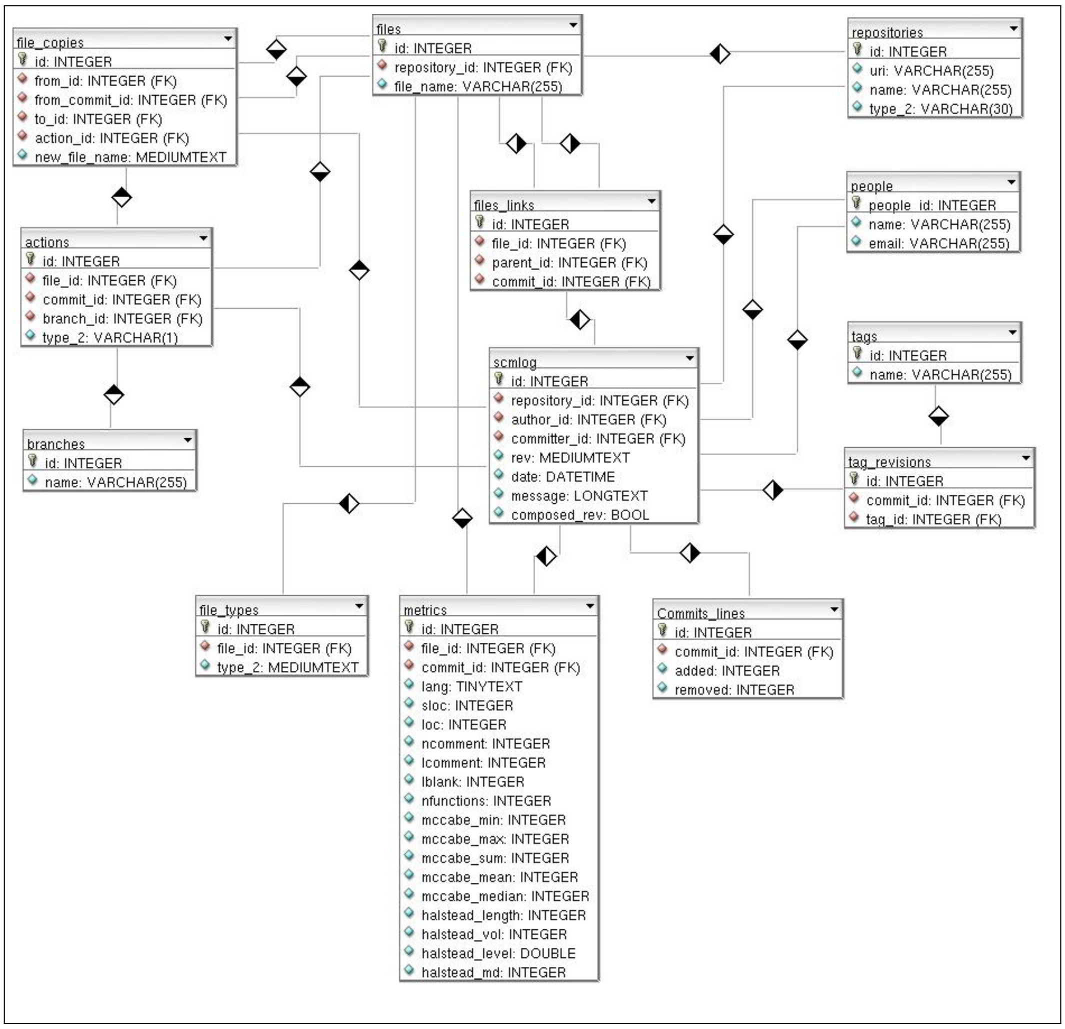
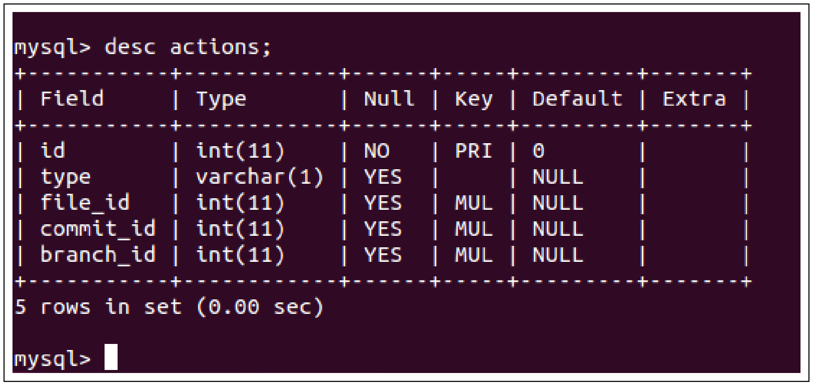


Table 3.2: CVSAnalY database schema



```
mysql> desc actions;
+-----------+------------+------+-----+---------+-------+
| Field     | Type       | Null | Key | Default | Extra |
+-----------+------------+------+-----+---------+-------+
| id        | int(11)    | NO   | PRI | 0       |       |
| type      | varchar(1) | YES  |     | NULL    |       |
| file_id   | int(11)    | YES  | MUL | NULL    |       |
| commit_id | int(11)    | YES  | MUL | NULL    |       |
| branch_id | int(11)    | YES  | MUL | NULL    |       |
+-----------+------------+------+-----+---------+-------+
5 rows in set (0.00 sec)

mysql> 
```

Table 3.3: CVSAnalY actions table

3.4 Software dependency metrics extraction

Measurement is a mapping of empirical objects to statistical objects with consideration given to all structures and relationships [59]. In this Thesis structural, logical and semantic dependency metrics are adopted and their relationships are empirically investigated in the context of OO and OSS. Software metrics are widely used in software engineering measurements to aid decision making, assess software quality and quantity (e.g., number of lines of code). The attributes measured by software metrics can be categorized into two groups: internal and external attributes [144]. The internal attributes of a software system include size, coupling or dependencies between artifacts and the amount of code reused in the system, while the external attributes include usability, reliability and security of a system [59]. In this research we are concerned with coupling of software components, thus we measure the internal attributes of the studied systems.

In Subsections 3.4.1, 3.4.2, and 3.4.3 the tools and techniques adopted to extract the logical, structural and semantic software dependencies between classes from the studied sample are described.

3.4.1 Logical coupling measurement

This task was a pure SQL extraction task, so it did not pose a time issue: for each project we extracted the number of revisions, based on the tables built by the CVSAnalY tool in a MySQL database. For each revision, we extracted the list of all possible pairs of classes that were co-changed in that revision.

The maximum number of classes co-changed in one revision was found to be 1,220 in one software project. Large commits have been often associated with a restructuring, or a bulk rewording of a software license[93]. Revisions with over 100 changed artifacts have been excluded from previous empirical studies and irrelevant co-changes have been filtered out of commits [149]. In their study on change impact analysis, Kagdi *et al.* discarded all non-source code files from commits, as their methodology was focused primarily on source code. In addition, commits with over 10 source files were also discarded [102]. This type of filtering is usually used when mining software repositories to reduce noise in logical coupling measurement, and to mitigate factors such as updating the license information on every file or performing merging and copying ([101, 210]. We have also adopted this approach in this Thesis.

Following the methodology in [102], we removed all files that did not have the `.java` extension and revisions without `.java` files since we are focusing on source code analysis [148]. In addition based on an interquartile range (IQR) analysis of the number of revisions per project in the sample of 380 projects, the projects with less than 20 (below first quartile – Q1) revisions were excluded [6]. That left us with a total of 79 open-source projects and this subset forms the studied sample of this Thesis. This final dataset and the source code of tools/scripts built to analyze it are shared in an open repository[7]. The projects are listed with brief descriptions from the source code authors in Section A.2.

There is a probability that attempting to prune before re-sampling will not

[6]This was suggested in the review received for a paper submitted to the empirical software engineering journal. A small number of revisions will not provide substantial information regarding logical coupling, for example in "projects with only two revisions one of which is the initial commit"

[7]https://figshare.com/projects/AN_EMPIRICAL_STUDY_ON_OBJECT-ORIENTED_SOFTWARE_DEPENDENCIES_LOGICAL_STRUCTURAL_AND_SEMANTIC/24466

yield a larger sample of non-trivial projects that meet the selection criteria because pruning before re-sampling could result in a larger number of trivial projects with less than 79 non-trivial projects left in the new sample. Previous studies have also excluded a number of OSS projects after their initial sampling. Samoladas *et al.* [171] and Gousios *et al.* [78] applied certain selection criteria to exclude projects from their initial selection. Midha and Palvia [139] based on certain project selection criteria, reduced their initial sample from 887 to 283.

Haefliger and Spaeth [79] reduced their selected sample of projects to 6 OSS projects with variance on their sampling criteria. The studied sample included a wide variety of software products such as office software, games, a hardware driver, and an instant messenger client and this reduced sampling bias [180]. Similarly in this study, the resulting non-trivial sample of 79 OSS projects are of different domains, sizes and levels of activity. The sample selection criteria widely used in OSS research [52, 161] and adopted by Haefliger and Spaeth, includes: 1) the project is under active development, allowing the tracking of its development activity, 2) the source code modifications of the project need to be available online, and 3) the project should have been in existence for at least a year.

An overall average of 6 classes co-evolving *per* revision was observed in the sample. After the extraction of pairs of co-evolving classes, we could then identify pairs of classes that frequently co-evolve or frequent association rules and compute logical coupling for each pair. According to Wiese *et al.*, "*change coupling is a phenomenon associated with recurrent co-changes found in the software history*" [196]. The historical information in software repositories "is an extension of the collective developer or development knowledge" [74].

Logical dependencies or evolutionary dependencies are based on the change history of two classes, and is a measurement of the observation those classes always co-evolve or change together [55, 66, 67, 195]. They are commonly treated as association rules [210], which means that when X_1 is changed, X_2 is also changed [148]. Furthermore, X1 and X2 are called the antecedent (also known as, left-hand-side, LHS) and the consequent (also known as, right-hand-side, RHS) of the rule, respectively. For example, the rule {A, B}→ C found that in the sales data of a supermarket indicates that a customer who buys A and B together, is also likely to buy C [148].

The logical coupling of two software artifacts are measured in terms of support and confidence [208]. The confidence value determines the strength of their association, in this case the degree or extent to which they co-evolve. Support is the count of times a pair of artifacts undergo co-change. These two metrics are further explained:

- *Support* – The support determines the number of revisions that two files are changed and committed together, in other words it is the number of changes made on file B while A is changed. E.g. if `GestionScore.java` was altered in 10 transactions (where 10 is the *"Transaction Count"* [203]. Of these 10 transactions, 8 also included changes of the file `Score.java`. Therefore, the support for the logical coupling of `GestionScore.java` ↔ Score.java is 8.

$$Support(A => B) = P(A \cup B)$$

- *Confidence* – The confidence determines the strength of the consequence of a given logical coupling, it is the support count / transaction count. E.g. If we assume that the file `GestionScore.java` was changed in total of 8 transactions, the confidence for `Score.java` → `GestionScore.java` is 8/8 = 1.0. In other words, we are 100% confident that `GestionScore.java` will be changed when `Score.java` is changed.

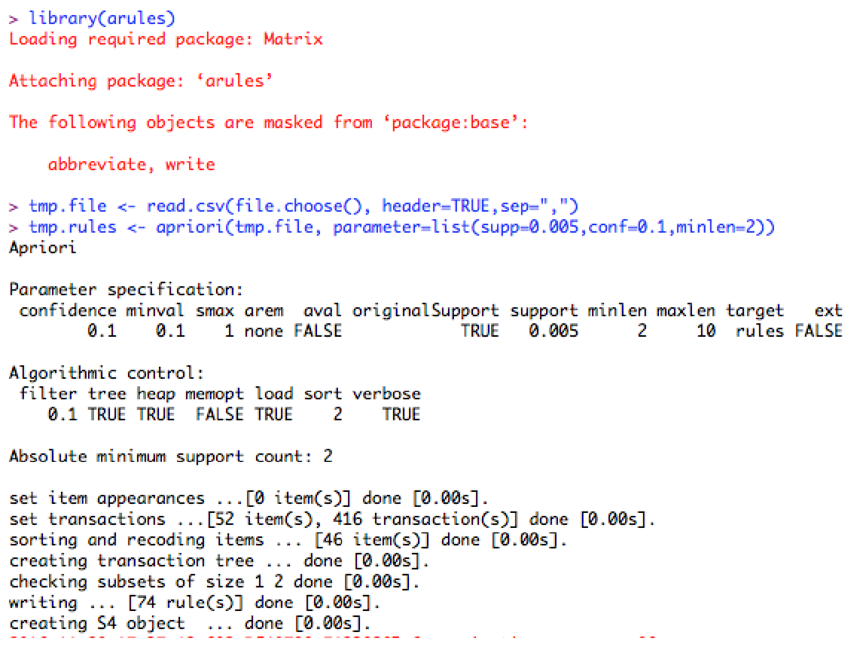
$$Confidence(A => B) = P(A|B) = \frac{P(A \cup B)}{P(A)}$$

Where P(A) is the probability of cases containing A. In this case revisions it is the revisions containing changes made to A.

This data can be computed using statistically evaluated tools such as WEKA [85] and the `arules` association rule mining library in R [81]. We have also adopted the `arules` package in the R statistical environment [81, 82, 206]. Canfora *et al.* [39] also adopted the *arules* package in computing logical coupling metrics in terms of support and confidence. In this Thesis we will focus more on the confidence of association rules. This is because we are more interested in the degree to which class pairs co-change in relation to structural and semantic coupling. In addition the support metric is only a count of the number of times two classes A and B are co-changed. While the confidence metric can be used to determine the probability of changing A if B is change or changing B if A is changed. Logical coupling measurement is not symmetric. In order words, the probability of changing B when A is changed is not the same as the probability of changing A when B is changed. Therefore, the confidence metric is adopted in this Thesis when evaluation logical coupling.

Likewise in previous research [108], the support and confidence thresholds have been set to 0.01 and 0.1, respectively. This is because increasing the support and confidence increases precision but lowers recall, thus identifying only a small number of association rules. The number of identified co-evolving sets reduces based on increase in confidence and such pruning looses important information [210]. Oliva and Gerosa classified confidence metrics as: [0.00-0.33] low logical coupling, [0.33-0.66] medium logical coupling and [0.66-1.00] high logical coupling and identified that highly logically coupled classes suffered slightest influence from structural coupling. In addition, the *arules* library in the R statistical environment has been used with a high precision and minimal false positives in prior research across different disciplines [80, 83, 108] when mining frequent item sets from data.

Figures 3.4 and 3.5 illustrate a computation of the logical coupling strengths of identified association rules in the *2dtetris* OSS project from our studied sample with the commands highlighted in blue. Each row consists of the antecedent (left hand side or lhs) and the consequent (right hand side or rhs) of a given rule, the support



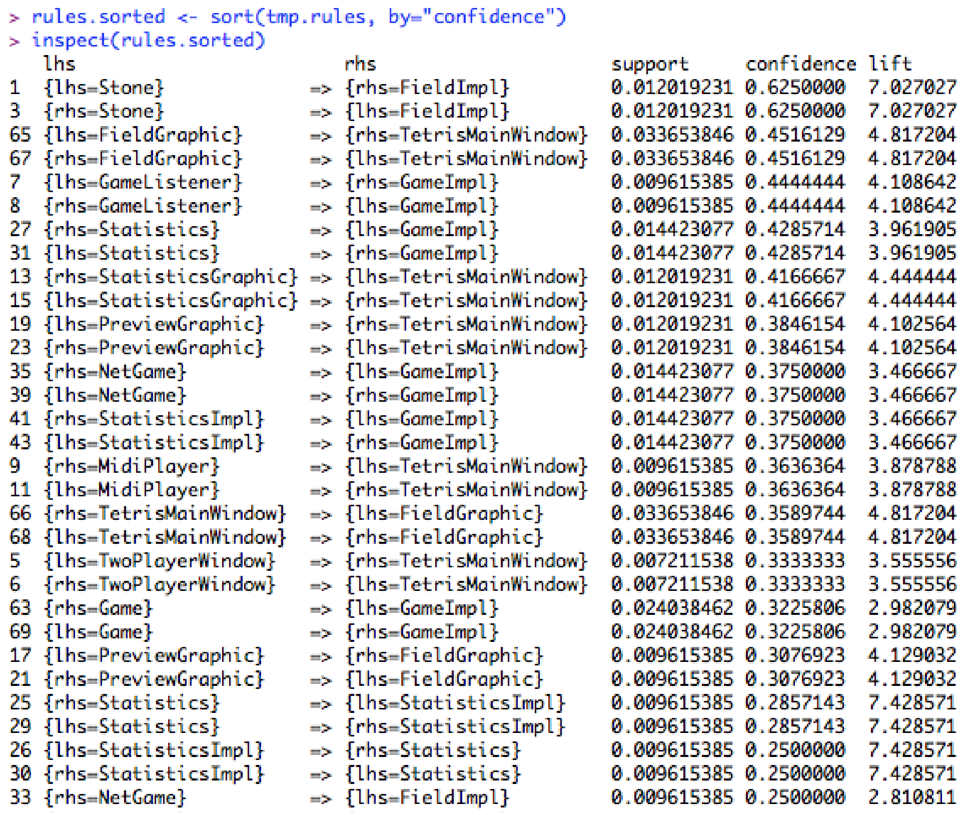
```
> library(arules)
Loading required package: Matrix

Attaching package: ‘arules’

The following objects are masked from ‘package:base’:

    abbreviate, write

> tmp.file <- read.csv(file.choose(), header=TRUE,sep=",")
> tmp.rules <- apriori(tmp.file, parameter=list(supp=0.005,conf=0.1,minlen=2))
Apriori

Parameter specification:
 confidence minval smax arem  aval originalSupport support minlen maxlen target   ext
        0.1    0.1    1 none FALSE            TRUE   0.005      2     10  rules FALSE

Algorithmic control:
 filter tree heap memopt load sort verbose
    0.1 TRUE TRUE  FALSE TRUE    2    TRUE

Absolute minimum support count: 2

set item appearances ...[0 item(s)] done [0.00s].
set transactions ...[52 item(s), 416 transaction(s)] done [0.00s].
sorting and recoding items ... [46 item(s)] done [0.00s].
creating transaction tree ... done [0.00s].
checking subsets of size 1 2 done [0.00s].
writing ... [74 rule(s)] done [0.00s].
creating S4 object  ... done [0.00s].
```

Figure 3.4: Association rule mining with the **arules** package in R

and confidence. The **arules** package provides a third metric which is the lift. This is a ratio of the actual confidence value and the expected confidence value to check whether each rule is listed by random chance [135]. A lift ratio greater than 1.0 means that the relationship between the antecedent and the consequent is more significant than would be expected if the two sets were independent. The larger the lift ratio, the more significant and interesting the association rule [135, 206].

$$Lift(A => B) = \frac{Confidence(A => B)}{P(B)} = \frac{P(A \cup B)}{P(A)P(B)}$$

In Figure 3.4, the first command loads the *arules* package, followed by a command which import the comma delimited (.csv) file (containing the data co-change history of Java class pairs in a software project) to be read. The third command used the **apriori**() function within the **arules** package which counts transactions to identify frequent itemsets and derive association rules from them. The following results after the last command shows that 74 association rules were identified out of which the top 30 sorted by the confidence metric are displayed in Figure 3.5.



```
> rules.sorted <- sort(tmp.rules, by="confidence")
> inspect(rules.sorted)
   lhs                          rhs                     support     confidence lift
1  {lhs=Stone}              => {rhs=FieldImpl}         0.012019231 0.6250000  7.027027
3  {rhs=Stone}              => {lhs=FieldImpl}         0.012019231 0.6250000  7.027027
65 {lhs=FieldGraphic}       => {rhs=TetrisMainWindow}  0.033653846 0.4516129  4.817204
67 {rhs=FieldGraphic}       => {lhs=TetrisMainWindow}  0.033653846 0.4516129  4.817204
7  {lhs=GameListener}       => {rhs=GameImpl}          0.009615385 0.4444444  4.108642
8  {rhs=GameListener}       => {lhs=GameImpl}          0.009615385 0.4444444  4.108642
27 {rhs=Statistics}         => {lhs=GameImpl}          0.014423077 0.4285714  3.961905
31 {lhs=Statistics}         => {rhs=GameImpl}          0.014423077 0.4285714  3.961905
13 {rhs=StatisticsGraphic} => {lhs=TetrisMainWindow}  0.012019231 0.4166667  4.444444
15 {lhs=StatisticsGraphic} => {rhs=TetrisMainWindow}  0.012019231 0.4166667  4.444444
19 {lhs=PreviewGraphic}     => {rhs=TetrisMainWindow}  0.012019231 0.3846154  4.102564
23 {rhs=PreviewGraphic}     => {lhs=TetrisMainWindow}  0.012019231 0.3846154  4.102564
35 {rhs=NetGame}            => {lhs=GameImpl}          0.014423077 0.3750000  3.466667
39 {lhs=NetGame}            => {rhs=GameImpl}          0.014423077 0.3750000  3.466667
41 {rhs=StatisticsImpl}     => {lhs=GameImpl}          0.014423077 0.3750000  3.466667
43 {lhs=StatisticsImpl}     => {rhs=GameImpl}          0.014423077 0.3750000  3.466667
9  {rhs=MidiPlayer}         => {lhs=TetrisMainWindow}  0.009615385 0.3636364  3.878788
11 {lhs=MidiPlayer}         => {rhs=TetrisMainWindow}  0.009615385 0.3636364  3.878788
66 {rhs=TetrisMainWindow}   => {lhs=FieldGraphic}      0.033653846 0.3589744  4.817204
68 {lhs=TetrisMainWindow}   => {rhs=FieldGraphic}      0.033653846 0.3589744  4.817204
5  {lhs=TwoPlayerWindow}    => {rhs=TetrisMainWindow}  0.007211538 0.3333333  3.555556
6  {rhs=TwoPlayerWindow}    => {lhs=TetrisMainWindow}  0.007211538 0.3333333  3.555556
63 {rhs=Game}               => {lhs=GameImpl}          0.024038462 0.3225806  2.982079
69 {lhs=Game}               => {rhs=GameImpl}          0.024038462 0.3225806  2.982079
17 {lhs=PreviewGraphic}     => {rhs=FieldGraphic}      0.009615385 0.3076923  4.129032
21 {rhs=PreviewGraphic}     => {lhs=FieldGraphic}      0.009615385 0.3076923  4.129032
25 {rhs=Statistics}         => {lhs=StatisticsImpl}    0.009615385 0.2857143  7.428571
29 {lhs=Statistics}         => {rhs=StatisticsImpl}    0.009615385 0.2857143  7.428571
26 {lhs=StatisticsImpl}     => {rhs=Statistics}        0.009615385 0.2500000  7.428571
30 {rhs=StatisticsImpl}     => {lhs=Statistics}        0.009615385 0.2500000  7.428571
33 {rhs=NetGame}            => {lhs=FieldImpl}         0.009615385 0.2500000  2.810811
```

Figure 3.5: Output of association rule mining with the `arules` package in R

3.4.2 Structural coupling measurement

The first study on software metrics was by Gilb [76] in which lines of code (LOC) was used to measure software quality and productivity. Chidamber and Kemerer [46] proposed static (structural) coupling metrics which have been widely used in software engineering (SE) research studies to measure software attributes[8]. Table 1 in Appendix A.1 summarizes the software dependency metrics proposed in the literature thus far based on the three dependency types studied in this Thesis.

While logical or evolutionary coupling computation is based on a time interval, structural coupling is defined for a specific time instant [71, 148] and is static. Geipel and Schweitzer [71] analyzed the link between structural dependency and co-change

[8]Such as coupling between objects (CBO), response for a class (RFC), lack of cohesion (LCOM), weighted methods per class (WMC) and sources lines of code (SLOC)

of classes. In calculating the linear relationship between structural dependency and co-evolution metrics they only take into consideration the latest code snapshot when extracting structural dependencies. However, they used either 0 or 1 to represent the absence or existence of a structural link between classes. They state that structural dependencies between two classes i and j are somewhat stable from the creation of the younger class until the removal of either i or j. Taking cues from Geipel and Schweitzer [71], we compute structural and semantic coupling metrics by analyzing the latest snapshot of the software projects in our sample.

From the last snapshot of the archived source code of each software project we retrieve the metrics for the strength of structural dependency between any pair of classes including; *"the number of external operational calls or references* (direct call relationships such as method calls, inheritance, or extension of another class, or a class aggregating objects of another class) [148] between "caller" and "called" classes, the number of methods making the calls *from* the "caller" *to* and the number of methods being called in the "called" classes using Scitools UNDERSTAND command line tool `und` [9] *via* multiple commands in a Perl script [125]. This phase of the methodology was very laborious, given the vast number of revisions to analyse: with an average of 2 minutes to extract the coupling data of a revision.

Oliva and Gerosa measured structural coupling using the Message Passing Coupling (MPC) metric. This is a measure of the number of external operation calls, i.e. the number of calls from methods of a class to operations of other classes. Yu [203] represented the reference (structural) coupling between classes with the dependency path count between two classes (dependency path is a path from the definition of the function `func (int i)` or variable `gv` in class C1 to the use of the function in class C2 [203]). Accordingly, the strength of the structural coupling from the *caller* C2 to the *called* class C1 in Figure 3.6 is 4 (2 for function call func(int), 2 for global variable gv) [203].

[9]https://scitools.com/

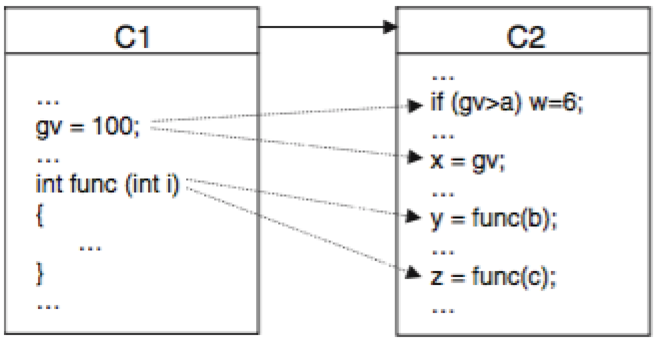


Figure 3.6: Structural dependency path between two OO software classes – *Caller* (C2) using a function and variable defined in *Called* (C1) [203]

3.4.3 Semantic coupling measurement

Prior research [102] has exposed the fact that semantic coupling between classes in OO systems is best measured at the file or class level of granularity. This is because the measurement of semantic coupling is more affected by the difference in granularity than logical coupling. For the semantic couplings, going from the coarse granularity of classes to the finer granularity of features or methods results in the reduction of the sizes of the documents or derived corpora. The documents are reduced in terms (and their frequency). That is, a corpus for a class is typically much “bigger” than a corpus for a method [102]. For logical coupling some commits do not contain changes made to methods while some do not contain changes made to classes, so there is no way to map changes made to classes and methods. For these reasons, we have chosen to study OO software at the class level of granularity in this Thesis.

Up till the time of carrying out this study, there has been a lack of adequate tools to automate the extraction of semantic coupling metrics based on the corpora of Java classes. The large size and complexity of today’s software systems necessitates the development of tools to effectively and efficiently help in automating program comprehension tasks [179]. The underlying mechanisms widely used in prior research to extract and analyze the semantic information from the source code are Latent Semantic Indexing (LSI) and the Vector space model (VSM).

3.4.3.1 Semantic coupling measurement using class corpora

The steps taken to extract and analyze the semantic information from source code as explained in previous work [155, 163, 187] and adapted by other studies are as follows:

1. Building the corpus: a corpus (“dictionary” of terms derived from comments, identifiers in source code) is built based on the level of granularity such that if a method level of granularity is chosen, after the source code is parsed – the extracted corpus (documents) will include method names and comments. At the class level of granularity, this will also include class identifiers and comments or documentation within the source code [74].

2. NLP: once the corpus is generated, it is preprocessed using natural language processing (NLP) techniques [74]. Mappings between methods, classes, and their indexes, respectively in the system corpus are generated in this step. Preprocessing of the system corpus is performed to eliminate common keywords, stop words, stem words, and to split identifiers [130]. In other words, all stop words (e.g., “a” and “the”) are removed, other words are stemmed (e.g., “fixes” to “fix”) [207].

3. Indexing the corpus: LSI (a variant of the vector space model) indexes the generated corpus and generates a real-valued vector description for each document [163]. In other words, LSI uses the corpus to create a term-by-document matrix (TDM), which captures the distribution of terms in methods (if extracted at the method level of granularity). The rows of the matrix correspond to the words from identifiers and comments and the columns represent the methods or classes. A cell m_{ij} in the TDM represents a measure of the importance of the i^{th} word in the j^{th} class or method [74]. Singular Value Decomposition (SVD) is then used then to construct a subspace, called the LSI subspace (or semantic space). Each document from the corpus (i.e., method from the

source code if extracted at the method level of granularity) is represented as a vector in the LSI subspace.

In contrast to the VSM approach, the method used by LSI to capture essential semantic information is dimensionality reduction; selecting the most important dimensions from a co-occurrence matrix (words by documents) decomposed using singular value decomposition (SVD) in order to exclude "inessential information" [163]. According to Revelle *et al.* "LSI was originally developed in the context of information retrieval as a way of overcoming problems with polysemy and synonymy that occurred with VSM approaches", e.g., the co-occurrence of terms like computer and laptop in the same document. It allows documents to be indexed by concepts rather than simple terms thus reducing noise. Some words occur in the same contexts, and a vital part of word usage patterns is dimmed by accidental and inessential information. In LSI the dimensionality of a corpus is the number of distinct topics represented in it. Notwithstanding, LSI has its own downsides [10].

4. Computing semantic similarities: Finally, once the corpus is indexed the cosine between two vectors is used as a measure of semantic similarity between two documents [96]. Just as cosine values range from −1 to 1, so do textual similarities. The closer a value is to one, the more similar the texts of the documents are [11].

   With regards to the vector space model (VSM), the similarity between any two documents is the cosine similarity between their corresponding vectors and can be computed as follows [96]:

$$CosineSimilarity(d1, d2) = \frac{Dotproduct(d1, d2)}{||d1|| * ||d2||}$$

[10]In LSI, the co-occurence matrix is decomposed by singular value decomposition (SVD), and its dimensionality is reduced by removing small singular values. Differences between VSM and LSI are discussed in Section A.3 in Appendix A

[11]http://www.ccs.neu.edu/home/jaa/CSG339.06F/Lectures/vector.pdf

Where the numerator represents the dot product (also known as the inner product) of the vectors $\vec{V}(d_1)$ and $\vec{V}(d_2)$, while the denominator is the product of their Euclidean lengths [12].

$$Dotproduct(d1, d2) = d1[0] * d2[0] + d1[1] * d2[1] * \ldots * d1[n] * d2[n]$$

$$||d1|| = squareroot(d1[0]^2 + d1[1]^2 + ... + d1[n]^2)$$

$$||d2|| = squareroot(d2[0]^2 + d2[1]^2 + ... + d2[n]^2)$$

Where d1 and d2 are n-dimensional vectors over the term set T = $t_0, \ldots,$ t_n . Each dimension represents a term with its weight in the document, which is non-negative [96]. Based on this, the cosine similarity is non-negative and bounded between [0,1].

The last two steps involving VSM and LSI, have been implemented using MATLAB[13] in previous research [155].

It is important to note that the semantic similarity between any two documents is symmetric. That is, the similarity between m_i and m_j is the same as the similarity between m_j and m_i. Therefore, the values of the semantic similarity between two methods are the same [102] irrespective of the order (`addShape()` $\leftrightarrow$ `removeShape()` and `removeShape()` $\leftrightarrow$ `addShape()`). It is noteworthy that this does not apply to structural or logical coupling measurements. For structural coupling, the number of calls or references from a class A to another class B are usually not the same in both directions. For logical coupling, the probability that a class A will be changed

[12]http://nlp.stanford.edu/IR-book/html/htmledition/dot-products-1.html

[13]MATLAB is a proprietary tool composed of a high-level technical computing language and interactive environment for algorithm development, data analysis and visualization http://uk.mathworks.com/products/matlab/.

when a change is made to B is not the same as the likelihood of changing B in response to a change in A.

Having outlined the steps above in computing semantic coupling between classes, certain gaps which have also been identified in Chapter 2 become obvious. Among the gaps are; (1) the process is convoluted with the adoption of different tools for different steps, (2) the absence of a tool to automate semantic coupling measurement by analyzing the corpora of `.java` classes in OO software and finally (3) analyzing the corpora of classes is a slower process compared to analysing only class identifiers. Especially when an OO software project is composed of thousands of classes and lines of code. In Subsection 3.4.3.2, we discuss a pilot study carried out to compare the measurement of semantic coupling using only the corpora of classes against their identifiers.

3.4.3.2 Semantic coupling measurement using class identifiers vs their corpora

In a prior pilot study [4], we have extended a tool written in Java[14] to automate the VSM approach to support the corpus-based measurement of semantic coupling at the class level of granularity in OO systems developed in the Java programming language[15]. The study was carried out on two software systems out of our overall sample to investigate whether the metrics derived from the corpora-based measurement echo those derived from using simple information retrieval techniques on the identifiers of the classes [4].

In this section, we outline the findings derived from extending the study on a larger subset of 35 projects out of our sample of projects to validate the results. Furthermore, based on the findings of the empirical comparison of measurement techniques, we will adopt the metrics derived from the most efficient measurement

[14]The extended tool supports step 1 in the corpus-based approach and was developed by the Software Engineering Maintenance and Evolution Research Unit at the College of William and Mary, Williamsburg, US. We contacted them and they gave us the permission to make use of their program written in Java

[15]https://github.com/najienka/SemanticSimilarityJava

technique in an investigation of the interplay between logical and semantic class couplings in Chapter 5 as well as the interplay between semantic and structural coupling in Chapter 6.

In the aforementioned pilot study [4], two sentence similarity measurement techniques namely N-Gram[16] and DISCO word synonym[17] category were compared against a corpus-based technique for calculating the semantic similarity between `.java` classes belonging to two OO software systems. The aim was to determine the feasibility of computing the semantic coupling of classes based on the similarity of their names rather than analyzing their content (source code). The results from the study extension are outlined in the following paragraphs.

To compare the corpora and identifier-based techniques, it is imperative to adopt a statistical test for independence or association to test for the association between the semantic coupling measurement techniques (identifier vs. corpora). Such a test will help establish the correlation between the metrics derived from the two types of semantic coupling calculation techniques (e.g., N-Gram vs. VSM). for this purpose, we adopted the Chi-square statistical independence test.

The first step is to populate a 2X2 contingency table, composed of row (i.e., *groups*) and column (i.e., *outcomes*) variables for each project. The first contingency table visible in Table 3.1 is a generic 2x2 contingency table, with the corpus-based outcomes (VSM) as the outcomes variable, and the identifier-based outcomes (N-Gram and Disco) as the groups variable. For the statistical test, three semantic dissimilarity thresholds $t = 0.1$, 0.2, and 0.5 used in previous studies [50, 51, 58, 109, 172, 183] on text similarity were selected and the correlation test results compared to identify the dissimilarity threshold at which there is a similarity in the semantic coupling metrics derivable from the measurement methods.

Where s is the semantic similarity between pairs, and using a semantic dissim-

[16]A Java implementation of the N-Gram distance algorithm is available at `https://github.com/tdebatty/java-string-similarity\#n-gram`

[17]The DISCO sentence similarity measures the semantic similarity between sentences according to the synonyms of their words. A Java implementation of the tool is publicly available at `https://sourceforge.net/projects/semantics/?source=directory`

ilarity threshold t (with a lower t implying a weaker similarity), the items of the contingency table are:

- A: pair of classes with $s \geq t$ for both Corpora-based and Identifier-based techniques;
- B: pair of classes with $s < t$ for one technique but $\geq t$ for the other;

The following are the possible outcomes observed for the threshold t – for the Identifier-based technique:

- C: pair of classes with $s \geq t$ for one technique but $< t$ for the other;
- D: pair of classes with $s < t$ for both techniques.

Table 3.1: Contingency tables: generic (top) and populated (middle and bottom) with identifier (either N-Gram or Disco) vs. corpus-based (VSM) techniques

Generic Contingency Table

	Corpora-Based (VSM)	
Identifier-Based	A	B
	C	D

VSM vs. N-Gram Comparison - **Geocoder-Java** project (p=.0104)

	≥ 0.1	< 0.1
≥ 0.1	208	8
< 0.1	439	47

VSM vs. Disco Comparison - **Geocoder-Java** project (p= < .0001)

	≥ 0.1	< 0.1
≥ 0.1	120	96
< 0.1	182	304

The other two tables (middle and bottom of Table 3.1) report the values and results for (i) VSM as the column variable, and N-Gram as the row variable and (ii) VSM as the column variable, and Disco as the row variable for a worked example – Geocoder-Java Project ($t = 0.1$).

Table 3.2: Geocoder-Java - summary of Chi-Square (contingency table) test results

Test ID	Project	Chi-Square Test (Technique A vs. B)	Semantic Dissimilarity Threshold t	p-value	Reject or Fail to Reject H_0?
1	Geocoder-Java	VSM $\leftrightarrow$ N-Gram	0.1	0.010	Reject
2		VSM $\leftrightarrow$ Disco		< 0.0001	Reject
3		VSM $\leftrightarrow$ N-Gram	0.2	< 0.0001	Reject
4		VSM $\leftrightarrow$ Disco		< 0.0001	Reject
5		VSM $\leftrightarrow$ N-Gram	0.5	< 0.0001	Reject
6		VSM $\leftrightarrow$ Disco		0.001	Reject

After populating the contingency Tables, we compute tests for association between the semantic similarity measures derived from the pairs of techniques (the identifier and corpus-based) using the Chi-square test method (chisq.test) in **R**[18]. This test is used to compare categorical data. It asserts the independence of the two techniques, with a null hypothesis H_0 of *no association between their outcomes*. We set the p-value at 0.01 as the threshold to reject the null hypothesis and compute the Chi-square tests for each project (99% confidence level).

Table 3.2 shows the p-values derived for the three semantic dissimilarity thresholds from the worked example – Geocoder-Java project. Based on the p-values in Table 3.2, we can see that the semantic similarity metrics derived from the identifier-based techniques reflect those derived from the corpora based technique at a dissimilarity threshold of 0.2. The p-values at this threshold are all below 0.01, thus we reject the null hypothesis at this dissimilarity threshold. Table 5 in Appendix C also shows that similarly to other projects in the sample, the identifier-based techniques (0.003 minutes) are faster in terms of computation time compared to the corpora-based (0.006 minutes) method for the Geocoder-Java project.

Analysing the rest of the projects, Figure 3.7 illustrates that at a dissimilarity threshold of 0.1, not all the p-values derived from the Chi-square test are less than or equal to 0.01. As a result, we cannot reject the null hypothesis. When the threshold is set to 0.2, an identical condition for 0.1 also applies for the VSM $\leftrightarrow$

[18]http://courses.statistics.com/software/R/Rchisq.htm

N-Gram tests with many outliers above the 0.01 mark. Yet the VSM ↔ Disco tests revealed a lower number of outliers while the rest of the p-values are less than or below 0.01. At a dissimilarity threshold of 0.5, the number of p-vlaues below the 0.01 mark are higher with only a few outliers especially in the VSM ↔ N-Gram tests. Thus there is substantial evidence to reject the null hypothesis using this dissimilarity threshold based on the Chi-square statistical test.

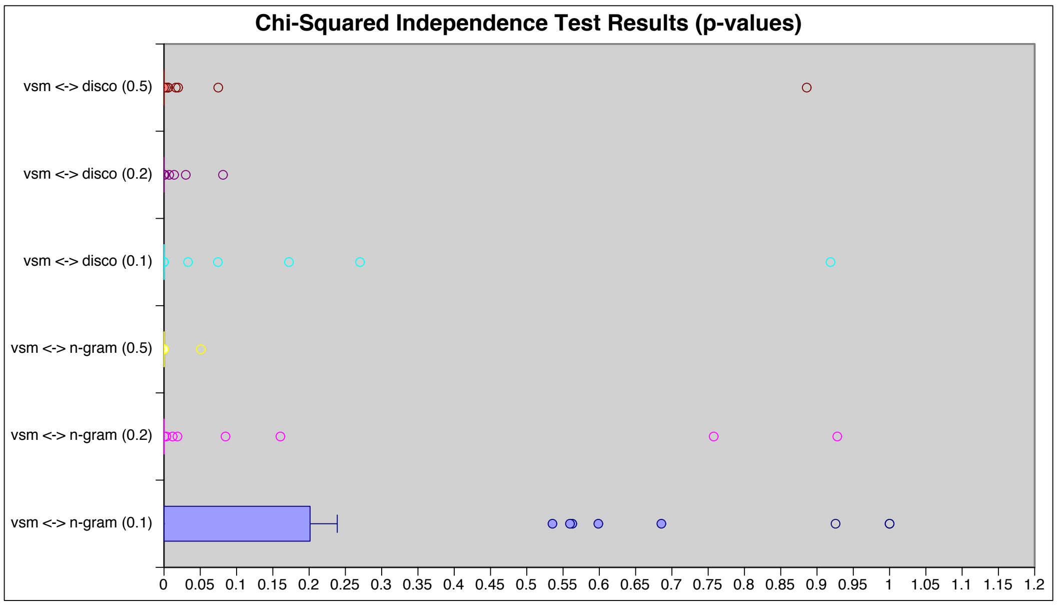


Figure 3.7: Chi-square association test results for class corpora (VSM) vs. identifier (N-Gram, Disco word synonym category) based semantic similarity techniques (box-plot distribution of p-values for threshold $t = 0.1$, 0.2 and 0.5)

Notwithstanding this feature, to further verify the independence test outcomes described in Section 5.4.1, in addition to the Chi-square test we compute the linear correlation between the corpora based semantic similarity measurement technique and the identifier-based techniques to verify whether the semantic coupling metrics reported by the different techniques for the same pairs of classes co-vary.

Spearman's rank correlation results indicated a moderate to large positive correlation (0.3 - 0.8) in at least half of the projects with a few outliers (negative correlation coefficients). Nevertheless, these negative correlation coefficients are statistically insignificant; the p-values are greater than 0.01 meaning the negative correlation is identified by chance.

Based on these findings there is not substantial evidence to reject the null hypothesis. This is because of the absence of a linear relationship but an association observed between identifier and corpora-based metrics (with Chi-squared tests). Thus, the conclusion is that semantic coupling metrics that leverage identifiers echo metrics derived when the whole class corpora are analyzed more often at a dissimilarity threshold of 0.5. The findings also prove that N-gram and Disco are much more computationally **efficient** than corpora-based techniques, time-wise.

The Disco technique compares words based on the similarities of their synonyms while the N-Gram technique is based on the edit distance and shared sub-strings of length n between sentences and has been widely used in the literature on text analysis [109]. We have used n-grams of size 4 in this study because research in the area of text mining [109, 134] has shown that n=4 maximizes precision when analyzing words or terms in several languages including English, French, German, Italian and Swedish. In addition, long lengths of n increase lexicon size, will not represent short words well and research has shown processing n-grams sizes larger than 10 is very slow [109].

The N-gram technique is more efficient than the Disco technique, precision-wise: the latter is heavily dependent on the English dictionary, as it considers words with similar English synonyms as semantically related. This study has shown that over 50% of the software projects analyzed do not contain classes with only English identifiers, therefore the Disco technique will produce many false negatives if used in the context of semantic coupling between OO software classes. In Chapters 5 and 6, we will leverage the N-Gram technique when computing semantic coupling.

While identifier-based techniques are more efficient when measuring semantic coupling between classes, the corpus-based technique is useful when recovering traceability links between source code and design documents [128, 199]. Identifier-based techniques are unable to extract the meaning or semantics of the documentation and source code to produce similarity measures that can be used to identify traceability links. This is because the identifier of a design document will be too vague and will likely be unrelated to a number of class identifiers. But when parts of the documents are parsed and compared with the terms embedded within the comments and source code of classes, then parts of design documents can be linked to classes in an OO software. Traceability is particularly useful when a developer is trying to comprehend someone else's code and following any provided documentation as is usually required during maintenance and evolution. This is usually done manually and can be time consuming (especially with large systems consisting of millions of lines of code) without tools that can automatically recover traceability links between source code and documentation.

3.5 Data analysis

Measurement is an indispensable approach in empirical software engineering. The findings of empirical studies are based on the values measured on research variables. Thus, the nature of measurements and their outcomes are important [192]. Statistics is useful for researchers to provide replicable and quantifiable conclusions from their research. According to Mubarak [144], "researchers are usually interested in measuring the relationship between two or more variables". These variables might be related in various ways; positively, not linked at all or negatively linked.

Table 1.3 in Chapter 1 summarizes the statistical tests adopted in this thesis. We have adopted the Spearman's rank correlation coefficient (ρ) for correlation analysis on the strengths or degree of class dependencies and the Fisher's exact independence test for investigating whether there is a statistically significant association between the class dependency types.

Spearman's correlation is a non-parametric test, i.e., it can be applied to data that is not normally distributed. We cannot guarantee that the strengths of the dependencies among class pairs in open-source software projects will be normally distributed across various revisions, neither can we guarantee the absence of outliers [164, 177]. Thus our rationale for applying non-parametric tests. The value of the correlation coefficient lies in the range $[-1; 1]$, where -1 indicates a strong negative correlation and 1 indicates a strong positive correlation. We adapt the categorization for correlation coefficients used in [129] ($[0 - 0.1]$ to be insignificant, $[0.1 - 0.3]$ low, $[0.3 - 0.5]$ moderate, $[0.5 - 0.7]$ large, $[0.7 - 0.9]$ very large, and $[0.9 - 1]$ almost perfect) if the rank correlation coefficient proves to be statistically significant at the $\alpha = 0.01$ level. Various correlation coefficients have been considered including Pearson, Kendall and Spearman. However, for Pearson's to be valid the data has to follow a normal distribution [151, 203] (the mean, median and mode have to be the same) while Kendall's tau is used in small sample sizes and where there are multiple values with the same score [62] and interpreted based on the probability of concordant and discordant observations. Finally, p-values derived from Kendall's tau are more accurate with smaller sample sizes.

The Fisher's exact independence test is adopted when comparing two categorical variables. It asserts the independence of the variables, with a null hypothesis H_0 of *no association between their outcomes*. We have used this test to analyze the association between measures derived from pairs of techniques for measuring semantic dependencies. This is to identify whether the measurements derived from applying one technique reflect those derived from a second technique. We have also adopted the test in investigating whether there is a significant association between pairs of class coupling types. The test performs better than the Chi-square X^2 test when there are empty cells in 2x2 contingency tables and we cannot strongly guarantee that all the cells in all contingency tables analyzed in this Thesis when comparing categorical variables will be populated. All the statistical tests are carried out using related functions in the RStudio[19] statistical environment.

[19]https://www.rstudio.com

3.6 Replicability

According to Lewis and Ritchie [116], "reliability is generally understood to concern the replicability of research findings and whether or not they would be repeated if another study, using the same or similar methods, was undertaken". Nahid [77] outlined three types of reliability with regards to quantitative research, as: (1) the degree to which a measurement, given repeatedly, remains the same (2) the stability of a measurement over time; and (3) the similarity of measurements within a given time period.

Hence, it is highly important for other researchers to be able to replicate and validate our study and findings as reliable. Based on this, we have ensured that the datasets and tools used in this study are publicly available and accessible. Three main empirical studies have been presented in this Thesis in Chapters 4, 5 and 6.

In Chapter 4, the interplay between structural and logical coupling is presented. When measuring logical coupling, the repository of each project was downloaded and stored, with its meta-data, using the CVSAnalY[20] tool[21]. The meta-data exposed the list of revisions for each class, and for each project as a whole. Each version of the project is then parsed using the `arules` association rule mining library in RStudio[22] to measure the confidence of the identified frequently co-changing classes.

In the structural coupling dataset, the coupling or dependency metrics measured are; *"the number of external operational calls or references* (direct call relationships such as method calls, inheritance, or extension of another class, or a class aggregating objects of another class) [148] between "caller" and "called" classes, the number of methods making the calls *from* the "caller" *to* and the number of methods being called in the "called" classes using Scitools UNDERSTAND command line tool `und`

[20]http://metricsgrimoire.github.io/CVSAnalY/

[21]Installation steps can be found at: https://sites.google.com/site/arnamoyswebsite/Welcome/updates-news/howtoinstallandruncvsanaly2inubuntu1110

[22]`https://www.rstudio.com`

[23] *via* multiple commands in a Perl script [125].

In Chapter 5, the interplay between logical and semantic coupling is investigated. In addition, identifier-based semantic coupling measurements are compared against a corpora-based method (Vector Space Model or VSM). While the identifier-based techniques have already been implemented and automated by other open-source software developers, the vector space model is automated by means of a tool we have developed in the Java programming language. This tool is one of our contributions and is further described in Section 3.6.1. The identifier-based techniques are; N-Gram[24] and DISCO word synonym[25]

In Chapter 6, we have studied the correlation between structural and semantic class dependencies and their overlap. The structural and semantic coupling metrics derived when carrying out the studies in Chapters 4 and 5 were adopted in this chapter.

The coupling datasets for the 79 open-source projects and the source code of tools/scripts built to analyze it are shared in an open repository[26]. The projects are listed with brief descriptions from the source code authors in Section A.2.

3.6.1 Corpora-based semantic coupling measurement tool

The tool we have developed for computing the semantic coupling of classes in Java projects based on their corpora can be publicly accessed on GitHub[27]. This tool is one of the contributions of this Thesis and has been outlined as part of our overall contributions in Chapters 1 and 7. As described in these chapters, this tool automates all the steps in Section 3.4.3.1. Previously, the last two steps had been carried out using Matlab which is distinct from the tool used to carry out the first

[23]https://scitools.com/

[24]A Java implementation of the N-Gram distance algorithm is available at `https://github.com/tdebatty/java-string-similarity\#n-gram`

[25]The DISCO sentence similarity measures the semantic similarity between sentences according to the synonyms of their words. A Java implementation of the tool is publicly available at `https://sourceforge.net/projects/semantics/?source=directory`

[26]`https://figshare.com/projects/AN_EMPIRICAL_STUDY_ON_OBJECT-ORIENTED_SOFTWARE_DEPENDENCIES_LOGICAL_STRUCTURAL_AND_SEMANTIC/24466`

[27]https://github.com/najienka/SemanticSimilarityJava

two steps.

Instructions on how to make use of the tool to compute the semantic coupling between classes can also be found in the GitHub repository[28] and are as follows:

- Copy the Java software project(s) folder(s) for which you want to compute the class level semantic similairty for and paste it in the Testcases/input folder. To test the tool, two Java software projects have been added to this folder in the repository.

- In your IDE (e.g. Eclipse), open the MainSemanticCoupling.java class (this is the main class and can be found in SemanticSimilarityJava/src/edu/wm/c-s/semeru/integration) and run it.

- This will identify the Java classes, build the corpus for each of them (eliminating common key/stop words, splitting words and stemming class and method identifiers) and will compute the semantic similarity between all possible class pairs ($\binom{n}{x}$, where n is the number of `.java` classes in the project and x = 2) using the Vector Space Model (VSM) technique.

- The output - the tool outputs the following: corpus for each class in your Java project, a .txt file containing the semantic similarity metrics between pairs of classes in each project will be saved in the home folder of this tool/project (SemanticSimilarityJava) and computation time in nano seconds for each project parsed (e.g., for the *ps3MediaServer* project, the tool outputs "time taken to compute VSM - 779562396021 nano seconds").

The important files in the project repository and created by the tool when parsing projects at the class level of granularity using the *ps3MediaServer* project as an example include:

- StopWords.txt - contains the stop words.

[28]https://github.com/najienka/SemanticSimilarityJava

- inputFileNamesps3mediaserver.txt - in the files folder, shows the names of the Java classes parsed.
- ps3mediaserverClassLevelSemanticCoupling.txt - in the VSM folder, shows the output of the semantic similarity between class pairs as computed based on the Vector Space Model (Cosine similarity).
- Corpus-ps3mediaserver-AfterSplitStop.corpusRawClassLevelGranularity - in the corpora folder, shows for each line the corpora of each class listed in the files folder. Other files in the corpora folder show the raw source code per line for each class, before pre-processing of the system corpus is performed to eliminate common keywords, stop words, split and to stem identifiers.

3.7 Summary of the chapter

In summary, we have outlined the set of techniques that have been adopted to conduct our empirical research; starting from the data extraction to the computation of static software class dependency metrics and statistical techniques adopted to explore the relationship between the software dependency types (i.e., logical, structural and semantic). To quantify the strength of the structural coupling from class to class, we extract the number of references (operational calls) between them. For logical coupling, we measure the confidence of frequently co-changed class pairs and on the semantic coupling axis, we measure the semantic similarity of class pairs. Both the logical and semantic coupling metrics range from 0 to 1.

Chapter 4 presents the first empirical study of this Thesis. The chapter will explore the interplay between the logical and structural coupling dimensions on the LSt axis in our sample of 79 OO OSS projects.

Chapter 4

Interplay on the logical and structural (LSt) coupling axis

4.1 Introduction

In the previous chapter, we explored the techniques that are applied in subsequent parts of the Thesis. To recap, the main aim of this Thesis is to understand the interplay between types of class dependencies in OO software; structural, logical and semantic dependencies. In this chapter, we investigate the interplay between the logical and structural coupling dimensions on the LSt axis.

Previous studies have provided contradicting results on distinct and limited samples of OSS projects. While some have stated that structural coupling always leads to co-evolution with both class dependency types sharing a linear relationship based on Spearman's rank correlation ρ [203], others have identified that a significant number of logical dependencies between classes cannot be accounted for by structural dependencies alone and *vice versa* [64, 71, 147, 148].

According to Oliva and Gerosa [147], controlling dependency levels in practice is still challenging. One of the reasons is that the way and the extent to which changes propagate by means of structural dependencies are still not clear. Learning from the inconsistencies and gaps in previous research, it is important to investigate the interplay between structural and logical coupling on a large scale and a completely distinct sample of OSS projects than those already investigated.

Structural signifies the set of class pairs with source code dependencies between them, while *Logical* signifies the set of classes observed to have co-changed in the past. In this chapter, we will lay emphasis specifically on the interplay between structural and logical dependencies between OO software classes. The rest of this chapter is structured in the following way. In the next section the motivation for this study is presented. For replicability, in Section 4.3, a description of the data collected and the investigated research questions are presented with a worked example in Section 4.4. Section 4.5 outlines the results for the overall sample of OSS projects while Section 4.6 discusses refactoring methods to resolve unnoticed class dependencies (e.g., class pairs linked logically but not structurally). Finally, a summary of the chapter in presented in Section 4.7.

4.2 Motivation

In Chapter 2 we defined the terms ‘structural coupling’ and ‘logical coupling’ and their operationalisation. In that chapter we also identified several gaps in the literature in relation to the interplay between structural and logical class dependencies in OO software. Researchers in this domain have produced contradicting findings, and as a result the interplay between structural and logical class dependencies is still less well-understood. Being better equipped with a firm finding and understanding of these dependency types has several impacts in software engineering research and software maintenance such as better prediction of software changes [1, 202, 209], focusing refactoring and testing efforts.

These contradicting findings need verification on a contrasting sample of OSS projects of different sizes. While it has been identified that structural coupling leads to logical coupling on a small sample of 12 classes written in C [203], other researchers have identified that most of the change dependencies have not occurred as a result of structural links between classes and *vice versa* [71, 147, 148].

The strength of the generalizeability of a previous research by Yu in which only 12 classes written in C were studied [203] to investigate the relationship between structural and logical coupling needs to be further improved with a larger sample of OSS projects [5]. Furthermore, the chosen p-value of 0.1 ($\alpha = 0.1$) could have resulted in a type I error – mistakenly rejecting a null hypothesis [49]. To reduce this threat, Yu planned to increase the confidence level to 95% (reducing the α value to 0.05) for more accuracy in future research. In a different study, they have estimated the CBO metric (the number of other classes structurally coupled to a class) [148] when computing structural coupling.

4.3 Empirical investigation

In this study, we will address the shortcomings of the previous research using a larger and contrasting sample of OSS projects. The main objective of this study is Obj1 described in Chapter 1 – investigating the interplay between structural and logical class dependencies in OO software. The premise is that the results will provide clearer and statistically verified findings on the interplay between structural and logical class dependencies in OO software.

The overarching null hypothesis H_0 to be verified in this study is: *There is no linear relationship between the strengths of logical and structural class dependencies in OO software.*

4.3.1 Studied sample of OSS projects

In Chapter 3, we outlined our project selection criteria and methodology. As a result of the criteria and methodology, 79 OSS projects formed our study sample. A brief description of each project is outlined in Section A.2 of Appendix A. Table

4.1 presents meta-data[1]. relating to these projects including an inter-quartile range analysis of the number of structurally coupled class pairs, logically coupled class pairs and the number of revisions *per* project. The lowest number of observed revisions in the studied sample of projects (21) was found in two projects *audao* and *ngamejava*; while the project with the highest observed number of revisions (769) is *ps3mediaserver*. The OSS project named *dbmigrate* has the lowest number of structurally coupled classes while *semanticdiscoverytoolkit* has the highest observed number of structurally coupled classes.

Table 4.1: Summary of project sample in terms of number of class dependencies and revisions.

	Min.	Q1	Median	Mean	Q3	Max.
Structural Dependencies	13	75	252	675	714	6,594
Logical Dependencies	26	394	1,648	21,640	10,441	529,590
Revisions	21	36	56	117	111	769

The box-plot in Figure 4.1 sums up the studied sample of OSS projects by the number of `.java` classes and revisions. The median number of revisions between 50 and 100. This is similar for the number of classes *per* project. As the upper outliers, *Semantic discovery toolkit*[2]) has some 1,500 classes, while *Ps3 media server*[3] underwent 769 revisions.

4.3.2 Data collected

OO software metrics have been used to measure quality attributes such as cohesion, coupling [8] and to predict fault proneness [3] as well as changeability of classes. In this study we are concerned with the strengths of the coupling of Java classes both structurally and logically.

In previous research, the structural and logical coupling strengths of classes is measured by the number of references between the classes [148, 203] and the confidence of their association rule [210]. These two metrics are at the core of

[1]Derived by conducting descriptive/summary statistics on the dependency metrics and number of revisions derived for each project using the **R Studio** statistics tool

[2]`https://github.com/git2020agile/semanticdiscoverytoolkit`

[3]`https://github.com/zrevai/ps3mediaserver`

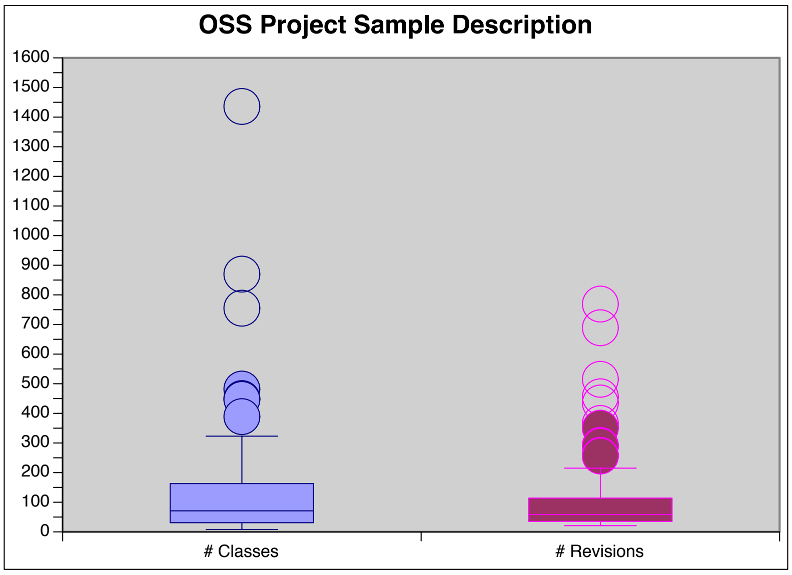


Figure 4.1: Summary of classes and revisions in overall studied sample of OSS projects

the computations and statistical analysis of this research. A description of the computation of the metrics from our studied sample of OSS projects is outlined in Chapter 3.

4.3.3 The research questions

This study is composed of two research questions (RQ1 and RQ2). These two main research questions to be answered are as follows:

- **RQ1:** Is there a large overlap between structural and logical coupling? This research question is based on the prevalence of results which state that the structural coupling of classes usually leads to their co-evolution [203]. Other studies have also mentioned that a majority of the logical or change couplings do not include structural dependencies and *vice versa* [71, 147, 148].

- **RQ2:** Is there a linear relationship between logical and structural coupling? This question is based on the premise that there is no linear relationship between the number of structural or operational calls between classes and the degree or strength of their co-evolution. This is because a previous study has identified a linear relationship with a high chance of false positives with a p-value of 0.1 [203].

4.4 Worked example

In this section, we analyse one OSS project from our sample of projects as an example in order to describe the approach taken in this study. The project under analysis is the *bluecove*[4] project (project ID = 28 in Table 2).

According to the project owners, "BlueCove is a JSR-82 J2SE implementation that currently interfaces with the Mac OS X, WIDCOMM, BlueSoleil and Microsoft Bluetooth stack found in Windows XP SP2 and newer. Originally developed by Intel Research and currently maintained by volunteers. BlueCove runs on any JVM starting from version 1.1 or newer on Windows Mobile, Windows XP and Windows Vista, Mac OS X. details". At the time of the data extraction for this study, the project had undergone 258 revisions and composed of 753 structural class dependencies, 63,404 logical class dependencies.

4.4.1 RQ1: Overlap between logical and structural coupling

To recap, RQ1 posed the research question "Is there a large overlap between structural and logical coupling?" Once pairwise structural and logical couplings were extracted *per* project, we parsed the data for each project to identify distinct structurally and logically coupled class pairs. After this we could then establish what percentage of structural couplings (number of references from *caller* to *called* classes > 0) includes logically related class pairs and what percentage of established logical couplings comprises of structurally related class pairs. This step is fundamental to answering RQ1.

For the *bluecove* OSS project we observed the following:

- 607 distinct class pairs belonged to the intersection set – both structurally

[4] `code.google.com/p/bluecove`

and logically coupled.

- 81% of structurally coupled class pairs were observed to have co-evolved frequently.
- ONLY 1% of co-evolved classes were structurally related (co-changed in one or more revisions).

This observation is further presented in Figure 4.2.

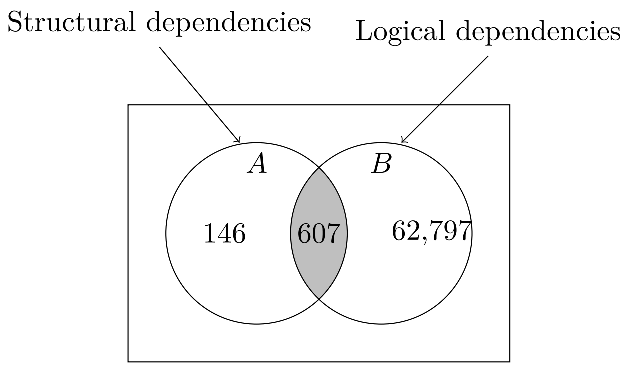


Figure 4.2: Intersection of structurally and logically coupled classes in *bluecove*

Several findings which can be drawn from these results, include:

- Structurally related classes usually undergo co-change. Table 3 in Appendix B shows that in this project, 81% of the structural class dependencies went through co-changes.
- Differently from [203], trying to infer structural coupling from co-evolution data will produce a lot of false positives [71].

4.4.2 RQ2: Linear relationship between logical and structural coupling

To answer the second research question ("Is there a linear relationship between structural and logical class dependencies?") we adopt the Spearman's rank correlation ρ described in Table 1.3 of Chapter 1. The input of the test is two vectors derived from the number of references between class pairs and the confidence of their co-evolution. As explained in Section 3.4.2, the references metric was extracted from the latest snapshot of the source code of each project in line with the state of the art [71] and reviews received for this study. The class pairs that form these vectors are those in the intersection set of RQ1 (a total of 607).

The null hypothesis H_0 as outlined in Section 4.3 to be tested is as follows:

- H_0: *There is no linear relationship between the strengths of logical and structural class dependencies in OO software.*

We have evaluated the correlation between the two vectors using the Spearman's rank correlation coefficient [203] because it is rare for either the structural or logical coupling values to have a normal distribution in each project.

We reject the null hypothesis at the 99% confidence level to minimise the chance of a type I error – mistakenly rejecting a null hypothesis. In other words, if the rank correlation coefficient proves to be statistically significant at the $\alpha <= 0.01$ level, we will reject the null hypothesis and fail to reject the alternative hypothesis H_1: *There is a linear relationship between the logical coupling and structural coupling of OO software classes.*

A threat to the statistical validity of [203] is the chosen confidence level (90%). In that study, the chosen $\alpha = 0.1$ could have resulted in a type I error. For more accuracy, the selected value is reduced in this research to mitigate this threat. Using the `cor()` function in the R statistical environment[5], we loop through the data for each OSS project in our sample and compute the Spearman's rank correlation to examine the linear relationship between the two coupling types being studied.

In the *bluecove* project, although co-changed more than once (in only a subset of revisions), the strength of co-evolution for all of the co-evolved classes was observed to be very weak (confidence metric = 0). This meant that there was no variance in the confidence metric (second vector) used in computing the correlation. As a

[5]http://www.statmethods.net/stats/correlations.html

result, the output of the correlation computation for both the correlation coefficient and the p-value was ‘NA’. This applies to other projects with a similar trend in our sample. Based on this result, there is no evidence of a correlation between the strengths of the structural and logical couplings between classes in the *bluecove* project.

4.5 Results: Overall sample of OSS projects

After illustrating the methodology adopted to answer both research questions (RQ1 and RQ2) in Section 4.4, we repeat the methodology with the overall sample of 79 studied OSS projects. The general results are forthwith presented in Sections 4.5.1 and 4.5.2. After this we discuss the findings, threats to the study and end the chapter with an overall summary.

4.5.1 RQ1: Overlap between logical and structural coupling

The core aim of this research question is to get a picture of the overlap of structural and logical class couplings in OO software projects by having a view of the proportion of distinct class pairs belonging to the intersection of the sets of structurally coupled and change coupled classes.

Figure 4.3 shows two summary box-plots with the following percentages:

$$CSD(\%) = \frac{Structural \cap Logical}{Structural} \tag{4.1}$$

$$CLD(\%) = \frac{Structural \cap Logical}{Logical} \tag{4.2}$$

The co-changed structural dependencies ratio (CSD) is the percentage of structurally coupled class pairs that have been observed to be usually modified at the same time, in the same revisions.

The coupled logical dependencies ratio (CLD) is the percentage of logically

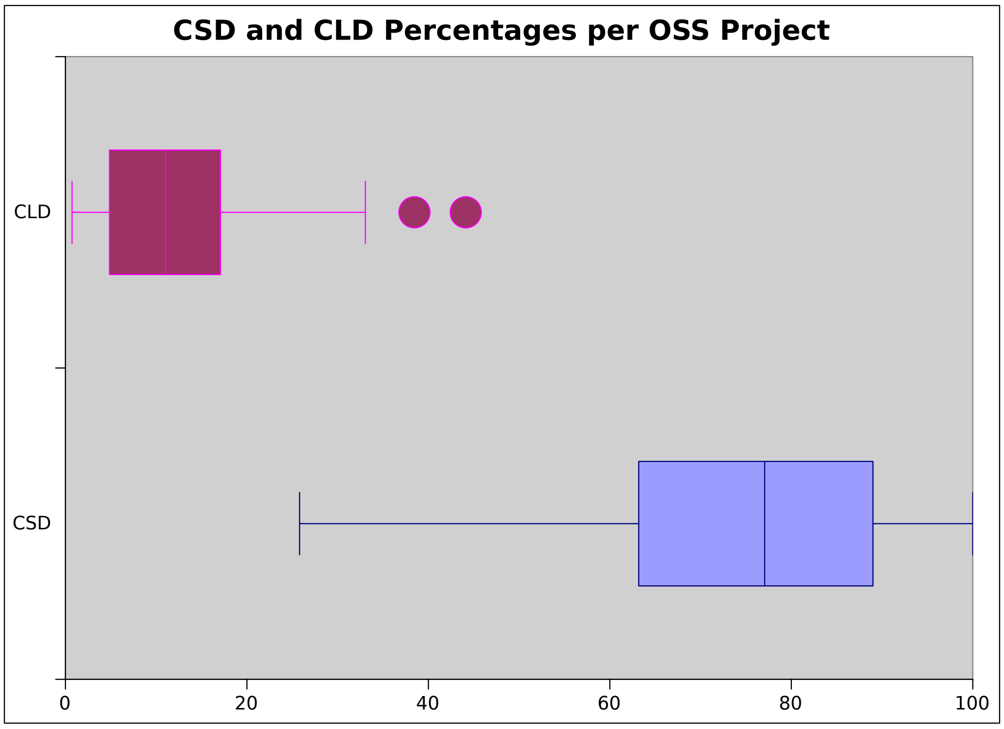


Figure 4.3: CLD and CSD Percentages *per* OSS project (KEY: CSD = co-changed structural dependencies; CLD = coupled logical dependencies)

coupled class pairs that have source code dependencies between them. The two box plots are exemplary of a common pattern: the median CSD ratio is around 80%, meaning that, for all the projects, a majority of the structurally coupled pairs also co-evolve. On the other hand, the median CLD shows that, for most projects, only a minority of the class pairs that co-evolve have source code links.

The 1st field in Table 3 in Appendix B shows the project IDs; the 2nd field shows the project names; the 3rd field shows the number of structural dependencies; the 4th field shows the number of logical dependencies; the 5th field shows the number of dependencies in the intersection set; the 6th field further shows the percentage of structural dependencies in the intersection set (CSD (%)); finally, the 7th field shows the percentage of logical dependencies in the intersection set (CLD (%)). The table is sorted by the 1st field in ascending order for better readability. Table 4.2 shows a subset (10 OSS projects) of Table 3.

Table 4.2: Intersection of structural and logical dependencies in a subset of 10 OSS projects. (KEY: Str. Dep. = structural dependencies; Log. Dep. = logical dependencies; CSD = co-changed structural dependencies; CLD = coupled logical dependencies)

ID	Project	Str. Dep.	Log. Dep.	Int. Set	CSD (%)	CLD (%)
12	alleywayreinvented	118	680	118	100	17
88	javacoder	16	104	16	100	15
97	jbandwidthlog	57	468	57	100	12
119	jsbe	23	70	23	100	33
189	sjava-logging	53	408	53	100	13
71	hobbylinkchecker	476	35,923	473	99	1
41	daedalum	252	4,854	249	99	5
136	migrator-postgresql	78	476	76	97	16
60	fyllgen	674	14,318	656	97	5
152	onslaught	297	5,739	289	97	5

For instance, we can infer from Table 3 that the OSS project *catchnthrow* with project ID = 31 has 68% of structurally coupled classes that include change dependencies. On the flip side, only 21% of the logical or change class dependencies include structurally coupled classes in the same project. This is a persistent trend in a majority of the other projects as is visible in Table 3. In 81% of the OSS projects in the studied sample of 79 projects, the results prove that structurally related classes usually consist of logically related classes (65 of 79 projects). In the remaining 14 projects, the percentage of co-changed structurally linked classes fall below 60%.

Two Venn diagrams Figure 4.4 (left and right) have been adopted to illustrate the two main observed trends in the results. The smaller circle represents the set of structural class dependencies while the larger circle represents the set of logical class dependencies. The top Venn diagram presents a scenario whereby ALL the structurally coupled class pairs in a project include logically or change related classes. These classes are observed to have always been co-changed instead of undergoing independent alterations. An example of this scenario is observed in the *jbandwidthlog* project (project ID = 97).

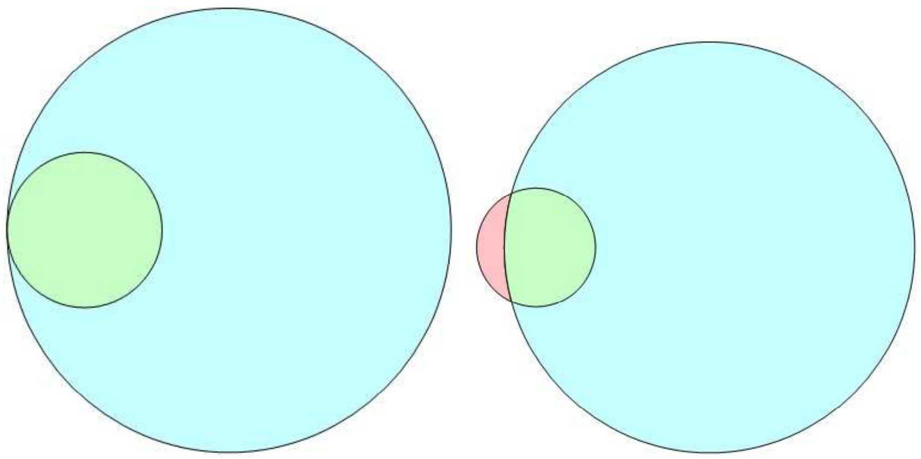


Figure 4.4: Venn Diagrams (weighted) showing the two sets of coupling in two scenarios: project ID=97 (left) and project ID=69 (right)

The second most common trend identified in the results is illustrated using the bottom Venn diagram in Figure 4.4 (right), showing the *guitarjava* project (project ID = 69). A subset of pairs of coupled classes do not need co-change, while the majority of the others still do. Again, in this project the majority of its other co-changes are not in favour of structural coupling.

In summary, a large proportion of the OO software projects have provided evidence to indicate that logically related classes do not usually involve structurally coupled classes. Furthermore, the proportion of logically related classes that involve structurally coupled classes OO software is very low in all projects studied. A previous study has shown that ripple effects are propagated across the path of source code links in OO software [147].

Geipel and Schweitzer [71] have demonstrated that the proportion of co-changed structural dependencies (CSD) are always larger than the proportion of structurally coupled logical dependencies (CLD) in open-source software projects. Similarly, based on a different sample of OSS projects, other researchers have found that the distribution of change dependencies is always larger than the distribution of structural dependencies [147, 148].

Differently from other research results, that tend to highlight the 80-20 Pareto distribution in most of the metrics on single software artifacts (complexity [45], defect density [60], number of changes [112]), the results from this study provide the evidence that structural dependencies in OO software projects do not follow such distributions.

Structural dependencies often include logical dependencies but not inversely

4.5.2 RQ2: Linear relationship between logical and structural coupling

Similarly to the worked example in Section 4.4.2, we have applied the Spearman's rank correlation method and the same OO structural and logical coupling metrics to answer RQ2 for the overall sample of OSS projects.

Using the Spearman's rank correlation, we tested for the null hypothesis H_0: *There is no linear relationship between the strengths of logical and structural class dependencies in OO software* (with $\alpha = 0.01$). Figure 6.3 shows the generic correlation outcomes, alongside the p-values derived from the Spearman's rank correlation analysis computation.

Correlation results are also outlined in Table 4 in Appendix B. As visible in the Table, a significant correlation (i.e., p-value $<= 0.01$) is not observed in any of the projects. In the majority of the projects; while some produced an insignificant (i.e., p-value > 0.01) negative Spearman's correlation between the structural coupling strength (i.e., references between classes) and their co-evolution strength (i.e., confidence), others produced an insignificant positive correlation coefficient [5]. Table 4.3 outlines results derived from a subset of 10 projects in the overall sample.

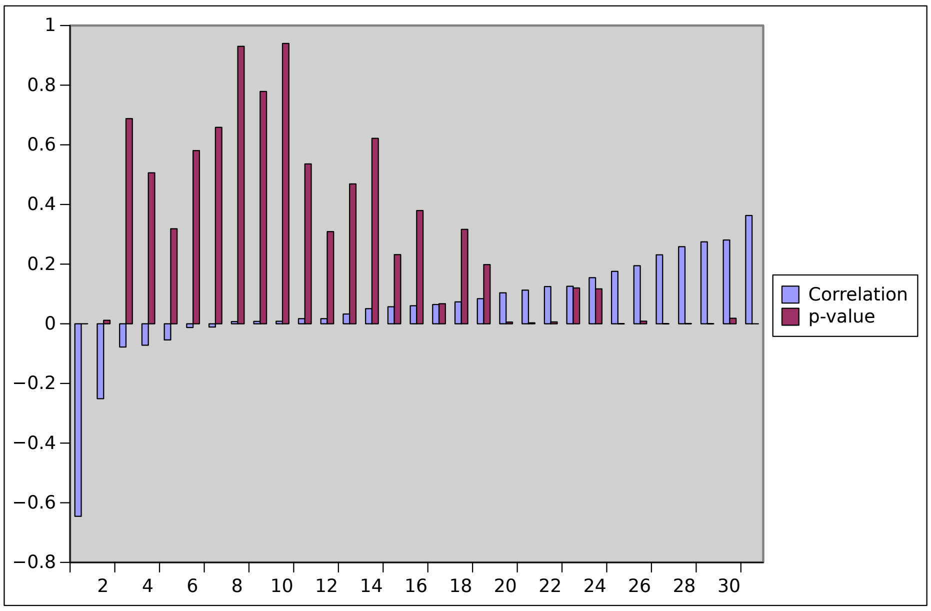


Figure 4.5: Structural and logical coupling - Spearman's rank correlation results with p-values

Table 4.3: Structural and logical coupling correlation outcomes of a subset of 10 OSS projects

ID	Project	Correlation Coefficient	p-value
1	2dtetris	0.1	0.4
2	4-connect	0.01	0.94
7	ahs-scheduling	0.01	0.93
14	amock	0.02	0.31
18	apjava	0.3	0
30	castanea	0.1	0.003
31	catchnthrow	0.2	0.01
41	daedalum	NA	NA
45	dbmigrate	-0.1	0.7
51	echo-nest-java-api	NA	NA

In a previous study, when using $\alpha <= 0.1$ and analyzing only a very small sample of twelve classes [203], it was inferred that there is a 'significant' correlation between the structural and the logical coupling of 12 classes of the Linux Kernel. On the contrary, using $\alpha <= 0.01$, this study has shown otherwise. From these results, one can infer that a stronger coupling between two classes is not a predictor of the likelihood of more changes to the same pair of classes. In summary, there is no solid evidence to reject H_0 presented in Section 4.4.2.

There is no significant correlation between structural and logical class coupling strengths; they have minimal impacts on each other

4.6 Points of action

The results presented in this chapter have statistically proven that not all classes that have previously co-changed are structurally linked. On the other hand, structurally coupled classes will usually lead to co-change. As such the overlap between structural and logical class dependencies is partially minimal because the relationship in one direction (structural $\rightarrow$ logical) is stronger than the other (logical $\rightarrow$ structural).

While structural coupling usually leads to co-change, not all co-changes will be identified during impact analysis solely based on an analysis of structural coupling. This implies that there could be other types of implicit dependencies linked to co-change such as semantic or conceptual coupling [102, 148].

Since not all the structurally coupled classes are been co-changed, one could infer that their structural links are somewhat stable or do not require modifications. As such, we cannot enforce a co-change between stable structurally related classes simply to maximize the overlap between structural and logical coupling as doing so can lead to unnecessary maintenance efforts. Therefore, the appropriate point of action will be to investigate the whether there is a significant relationship between semantic and logical dependencies.

4.7 Summary of the chapter

In this chapter, we have investigated the interplay on the LSt axis by focusing on the interplay between structural and logical class dependencies in 79 OO software projects. Firstly, we investigated the overlap between both coupling types by quantifying the proportion of class pairs belonging to the intersection set (both source code and change related class pairs) *per* project. Results showed that co-evolving classes are not always related by operational calls. However, a majority of source code related classes will undergo co-change. Secondly, adopting Spearman's rank correlation which is a statistical test used to probe the linear relationship between two variables, we found no strong evidence in favour of a linear relationship between structural and logical coupling strengths. This means that a stronger structural coupling strength does not imply a higher chance of co-evolution.

These results are useful for the software engineering community and have several impacts in software engineering and maintenance: majority of co-changes cannot be accounted for by source code links therefore, structural class dependencies will usually include logical dependencies therefore, coupled classes should be co-tested after modifications to a class to mitigate the chance of future faults.

Investigating the interplay between semantic and logical coupling will be a fertile research topic. The rationale is the possibility of accounting for co-changes to classes by means of their semantic similarities. Another rationale is the use of semantic coupling metrics to inform practitioners about the strength of the co-evolution of class pairs without the need to analyze historical data which in some cases might simply be unavailable. In Chapter 5, the interplay on between logical and semantic coupling dimensions on the LSe axis.

Chapter 5

Interplay on the logical and semantic (LSe) coupling axis

5.1 Introduction

In the previous chapter, an investigation into the interplay between structural and logical class dependencies in OO software was carried out. The goal of that study was to investigate the presence of a linear relationship between structural and logical coupling, as well as a directional relationship identified in prior studies on distinct samples of OSS projects.

To recap, one of the highlights of Chapter 4 is structural dependencies between OO software classes give rise to their co-evolution. However, not all the logical class dependencies could be accounted for by structural dependencies. The implication of that is most of the logical dependencies between classes in the analyzed sample of OSS projects could have been brought about by means of other types of dependencies (for example, semantic coupling). Another lesson learned from Chapter 4 is that one cannot assume that there is a structural link between two classes based on a high co-evolution degree.

Semantic coupling is based on the textual similarities between the terms embedded within the source code of OO software classes, including their comments and identifiers while *logical coupling* is based on an observation of the historical co-evolution of classes. In this chapter we will lay emphasis specifically on the interplay between the logical and semantic coupling dimensions on the LSe axis.

The rest of this chapter is structured in the following way. In the next section the motivation for this study is presented. For replicability, in Section 5.3 a description of the data collected and the investigated research questions are presented with a worked example in Section 5.4. Section 5.5 outlines the results for the overall sample of OSS projects. In Section 5.6 we discuss points of action in scenarios whereby one coupling type does not reflect another (e.g., classes linked semantically do not share a logical link). This is followed by a summary of the chapter in Section 5.7.

5.2 Motivation

In a prior study, Yu [203] has identified that there is a linear relationship between structural and logical coupling and that structural coupling leads to co-evolution though on a small sample of C classes. No large scale empirical study has been carried out to understand the interplay between logical and semantic class dependencies. Oliva and Gerosa have identified this to be a fertile research topic [148]. This is because the structural coupling of classes could not account for all the logical dependencies between classes. These gaps form the motivation of this study and we propose the premise that a large proportion of semantic class dependencies in each OSS project in the studied sample of software projects will consist of logical dependencies [21]. Figure 5.1 embeds this premise (highlighted in orange) in the body of prior knowledge on class dependencies.

According to Bavota *et al.* [21], "the peculiarity of the semantic coupling measure allows it to better estimate the mental model of developers than the other coupling measures. This is because, in several cases, the interactions between classes are encapsulated in the source code vocabulary". Therefore, it is important to leverage

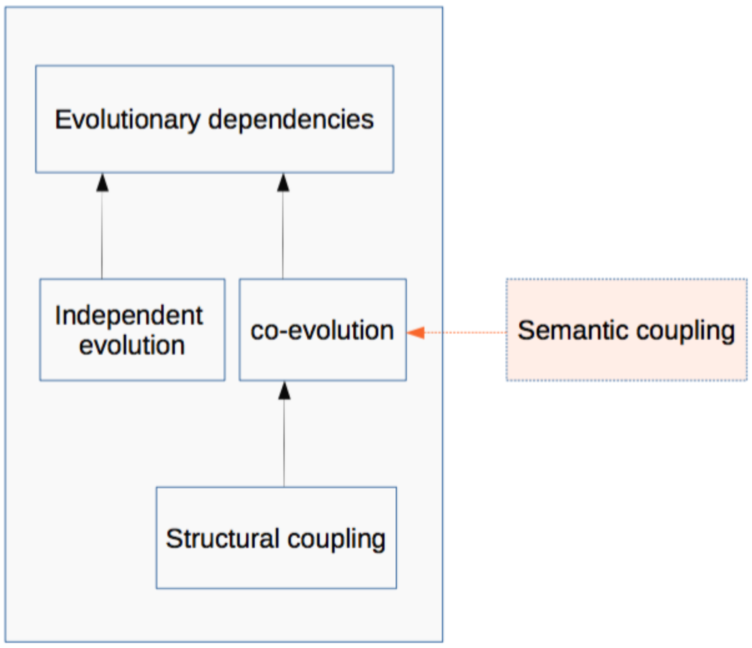


Figure 5.1: Motivating example: structural coupling → co-evolution [203] and semantic coupling → co-evolution (premise)

the information embedded in the identifiers and comments of classes which are not captured by an analysis of structural dependency when carrying out static change impact analysis [156]. This is because developers are likely to use similar terms and comments in related classes to describe the shared features among classes. This study aims to address the aforementioned gaps.

5.3 Empirical investigation

The main objective of this study is Obj2 described in Chapter 1: to investigate the interplay between semantic and logical coupling among classes in OO software. In this section, we describe the study methodology with worked examples.

5.3.1 Studied sample of OSS projects

The overall sample of OSS projects analysed in this study are described in Section 3.4.1 and Table 4.1. We have also described the project sampling techniques in Chapter 3. To briefly recap, the lowest number of observed revisions in the studied sample of projects (21) was found in two projects *audao* and *ngamejava*; the project with the highest observed number of revisions (769) is *ps3mediaserver*. The same project sample has been studied throughout this Thesis.

5.3.2 Data collected

OO software metrics are used to measure the quality attributes of software such as coupling or dependencies of components [35, 198], fault proneness [3] and cohesion [30, 97]. This study is primarily focused on the semantic and logical coupling of classes in OO software projects. In Chapter 2 we described semantic and logical coupling of OO software classes and explained the computation of associated metrics in Chapter 3.

To recap, semantic coupling [155] is a measure of the degree of the semantic similarity of the terms embedded in the source code, comments and identifiers of classes, while the confidence metric [210][1] of identified association rules or frequently co-changing classes is a measure of the degree of their co-evolution. Both metrics lie between the range of 0 and 1, where 0 is the minimum and 1 is the maximum. This metric pair are adopted in the statistical analyses of this chapter. We also determined in Chapter 3 that the N-Gram sentence similarity technique performs better on English and non-English terms. Therefore, semantic coupling metrics in this study are based on the N-Gram sentence similarity technique.

5.3.3 The research questions

Based on the identified gaps and motivations, several research questions were conceived to be answered in this study. Thus, this study aims to answer the following

[1]for logical coupling

research questions:

- RQ1: Is there a large overlap between semantic and logical coupling?

  This question is based on established results from previous research on the link between structural and logical coupling. It is also important to understand the degree to which semantic coupling influences co-change of classes in OO software projects.

- RQ2: Is there a linear relationship between semantic and logical coupling?

  Answering this question will shed more light on the extent to which semantic coupling between classes influences the degree of their co-evolution. Previous research has established a linear correlation between structural and logical coupling strengths. For the sake of change impact analysis, it is also imperative to have an understanding of the degree to which semantic similarity of classes and the strength of their co-evolution co-vary.

5.4 Worked example project

For the purpose of replicability we describe the study methodology using the *Geocoder-Java* OSS project as an example from our sample. The project is the third version of an implementation of a Java API for Google geocoder.

5.4.1 RQ1: Overlap between logical and semantic coupling

In order to answer RQ1, it is essential to identify pair-wise semantic and logical class dependencies. Once the class dependencies are identified, a spreadsheet is populated with the following columns; LHS (antecedent), RHS (consequent), semantic similarity, and confidence. To recap, as described in Chapter 3 the antecedent and consequent are derived from mining association rules in the evolution history of classes. Based on frequently co-changing classes, the consequent is a class that is usually changed when a change is made to the antecedent and the strength of this co-change is the confidence metric [210].

After pair-wise class dependencies are identified with associated semantic and logical dependency metrics, we built the aforementioned spreadsheet. Using a Shell script, we parsed the data and identified the proportion of semantic dependencies that involve non-logical dependencies (i.e., $A - B$ from the sets in Figure 5.2), the proportion of logical dependencies that involved non-semantic dependencies (i.e., $B - A$ from Figure 5.2) as well as the intersection set of pairs of classes that are both semantically and logically related (i.e., $A \cap B$ from Figure 5.2). In an example OSS project *Geocoder-Java*, we identified 441 semantic dependencies, 379 logical dependencies and 379 class pairs in the intersection set. This means that in this project, 100% of the logical dependencies include classes with semantic relationships. On the other hand, not all the semantically related classes were co-changed.

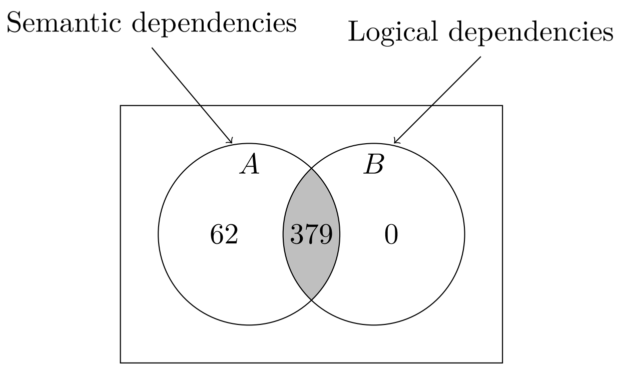


Figure 5.2: Intersection of semantically and logically coupled classes in *Geocoder-Java*

5.4.2 RQ2: Linear relationship between logical and semantic coupling

In this section, we describe for the sake of replications the calculation of the statistical test for RQ2. To recap, RQ2 poses the research question “Is there a linear relationship between semantic and logical coupling?”.

We created two vectors, one with the values of the logical coupling strength (i.e., the *confidence* metrics) and the second one with the values of the semantic coupling strength *per* pair of classes, and along their string of revisions.

The observations in the two vectors include the semantic coupling metric and the confidence metric of the co-evolution of two classes. The semantic coupling metric is a symmetric metric meaning that the semantic coupling is the same in both directions (for example a pair of classes i and j in a software project Y, $i \rightarrow j$ will have the same semantic coupling metric as $j \rightarrow i$). The logical coupling metric however is not symmetric. The association rule $i \rightarrow j$ measures the strength of the following observation: "when i is modified, there will always be a change in j" [208]. Therefore, $i \rightarrow j$ and $j \rightarrow i$ are not treated as the same association rule (the confidence metric could be different but the semantic coupling metrics is the same) and are considered as different observations in the created vectors. Both vectors will be composed of the same number of observations with the confidence metric in one and the semantic coupling metric in the other.

The null hypothesis H_0 to be tested is as follows:

- H_0: *There is no linear relationship between the strengths of logical and semantic dependencies.*

The correlation between the two vectors is evaluated using the Spearman's rank correlation coefficient [203]. In this study we calculated the Spearman's rank correlation coefficients using the built-in `cor()` fucntion in the **R** statistical environment (example usage in R, `cor.test(vector1, vector2, method = "spearman"` [2])). As described in Chapter 3, Spearman's metric (non-parametric) was selected over other correlation techniques because we cannot guarantee that either the semantic or logical coupling metrics will follow a normal distribution. We reject the null hypothesis for all studied projects at the 99% confidence level.

Simply put, if the calculated Spearman's rank correlation coefficient proves to be

[2] `http://www.gardenersown.co.uk/education/lectures/r/correl.htm#cor_sig`

statistically significant at the $\alpha = 0.01$ level, we will reject the null hypothesis and fail to reject the alternative hypothesis H_1: *There is a linear relationship between the semantic coupling and logical coupling of OO software classes.* The results derived for all projects are described in Section 5.5.2. The $\alpha = 0.01$ level was chosen as suggested in Yu's study [203]. One of the threats to the statistical validity to their study was the selection of the significance level. In that study, they chose $\alpha = 0.1$ which might have resulted in a type I error - mistakenly rejecting a null hypothesis. To reduce this threat, they planned in future research to increase the α value to 0.05 for more accuracy (which we have done herein with $\alpha = 0.01$). In project *Geocoder-Java*, the calculated Spearman's rank correlation coefficient of 0.06 is insignificant with a p-value of 0.21.

5.5 Results: Overall sample of OSS projects

Applying identical research methodology and statistical tests described in Section 5.4 for the worked example OSS project on the overall sample of 79 OSS projects, we describe the results in this section.

5.5.1 RQ1: Overlap between logical and semantic coupling

With the goal of contributing to the body of knowledge on the interplay between semantic class dependencies and their co-evolution, we further empirically probed the absence or presence of a large overlap between semantic and logical coupling. This is to understand the effect of semantic coupling on co-change and *vice versa* if any. In Chapter 3, we have discussed with empirical results, the association between class identifier and corpus-based semantic similarity calculation techniques at a semantic dissimilarity threshold of 0.1. We also identified that the N-Gram sentence similarity used in calculating class identifier similarity performed better on non-English terms. Accordingly, in this section the semantic coupling between classes is calculated based on their identifiers only and we have considered class pairs to be semantically related if their semantic similarity is between 0.1 and 1. In order to answer RQ1, it is imperative to gain an understanding of the intersection of the logical and semantic coupling *per* project. This intersection set is depicted by the shaded area in Figure 5.2. This set of classes *per* project is defined as *the proportion of class pairs linked both logically and semantically.*

Table 6 in Section C.1 shows for each OSS project in our sample of 79 projects, the number of distinct semantic dependencies in the third column, the number of distinct logical dependencies in the fourth column, the number of dependencies in the intersection set – pairs of classes that co-change and are semantically related, the percentage of semantic dependencies in the intersection set shown in the sixth column (see equation 5.1), while the last column shows the proportion of logical dependencies in the intersection set (see equation 5.2).

Equations 5.1 and 5.2 are at the core of RQ1. Two formulas are presented: the Co-changed Semantic Dependencies (CSD, measured in percentages) and Semantic Logical Dependencies (SLD, also a percentage). These two formulas are used as a measure of the class dependencies that belong to the intersection set (both logically and semantically related classes). The CSD(%) represents co-changed semantic dependencies; these are class pairs that share a semantic and modification relationship (frequently co-changed). The SLD(%) represents classes that are logically or change related and also share a semantic relationship. Some classes might only share either a semantic relationship only or a logical relationship only and these classes do not belong to the intersection set.

Figures 5.3 exhibits two summary plots with the following percentages:

$$CSD(\%) = \frac{Semantic \cap Logical}{Semantic} \tag{5.1}$$

$$SLD(\%) = \frac{Semantic \cap Logical}{Logical} \tag{5.2}$$

Table 6 in Appendix C is sorted by the project IDs and names for readability.

The table shows that there is a directional connection between co-change and semantic coupling. When the identifiers of pairs of classes contain similar terms (i.e., the classes are semantically related) they usually require modifications at the same time. This also holds in the opposite direction.

The results mentioned above are illustrated with two box-plots each in Figure 5.3. The plots show the distribution of class pairs belonging to the intersection set (classes with both semantic and logical dependencies; see equations 5.1 and 5.2). The results indicate a bi-directional relationship between semantic relationships and co-change, especially in Figure 5.3 where both distributions are relatively high in the overall sample of studied OSS projects. Therefore, we reject the null hypothesis (H_0) for RQ1 presented in Table 1.2 (Chapter 1) and fail to reject the alternative hypothesis: *There is a directional relationship between the semantic and logical dependencies among OO software classes.*

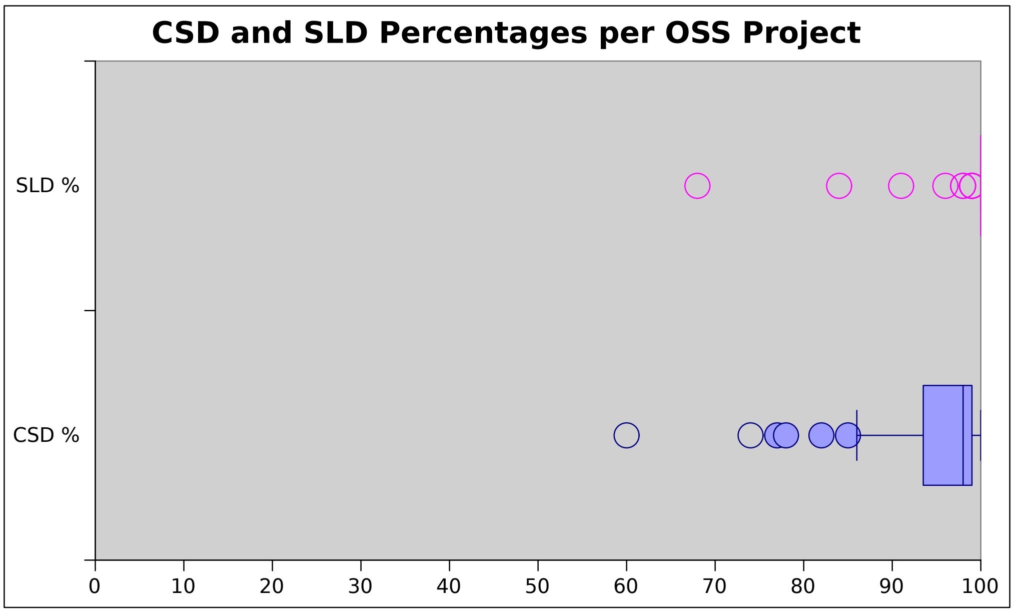


Figure 5.3: CSD and SLD percentages *per* OSS project (KEY: CSD = co-changed semantic Dependencies; SLD = semantic logical dependencies; semantic dependencies calculated with N-Gram identifier-based technique)

Semantic dependencies often include logical dependencies and this also applies inversely for identifier-based semantic similarity technique not sensitive to only English terms

5.5.2 RQ2: Linear relationship between logical and semantic coupling

Results in Section 5.5.1 have shown that commonly, semantic class dependencies will include logical class dependencies in OO software projects. In this section, following the methodology in Section 5.4.2 we aim to answer RQ2 with an investigation of the overall sample of 79 OSS projects. This is to determine the effect of the strengths of the semantic and logical coupling of class pairs on one another. The strength of the logical coupling is measured in terms of the confidence metrics of identified association rules or frequently co-changed class pairs. This metric ranges between 0 and 1. Previous studies on the relationship between the class dependency types have not answered RQ2 or investigated whether a high semantic similarity will lead to a high degree of the co-evolution of class pairs. In answering RQ2, we test for the following null hypothesis:

- H_0: *There is no linear relationship between the strengths of logical and semantic dependencies.*

The measurement of how loosely or closely two classes are semantically coupled is based on the class identifier-based method (N-gram) described in Chapter 3.4.3. The reason for this is that we have identified the computational efficiency of the identifier-based methods compared to the corpora based method in Section 3 as well as a linear relationship between their outputs (semantic coupling measurements). The linear correlation between both metrics is investigated and the results are now presented. Answering RQ2 will shed more light on whether or not the strengths of the semantic and logical (change) coupling of OO software classes co-vary. For instance, if they do co-vary, such statistical results will enable the prediction of the co-change frequency of class pairs based on the strength of their semantic coupling.

Adopting the Spearman's rank statistical test for correlation (ρ), we created two vectors as its input *per* studied OSS project. The first one with the semantic coupling metrics of class pairs and the second one with the confidence metrics, a measure of the co-evolution degree of frequently co-changing class pairs.

Figure 5.4 shows the correlation outcomes along with p-values obtained for the identifier-based semantic similarity measurement techniques. The underlying Spearman's rank correlation data is also presented in Table 7 in Appendix C. Figure 5.5 gives a clearer picture of the distribution of the p-values. Identically to the Chi-squared test for independence explained in Section 5.5.1, we reject the null hypothesis at the 99% confidence level with only a 1% error margin for the Spearman's rank correlation.

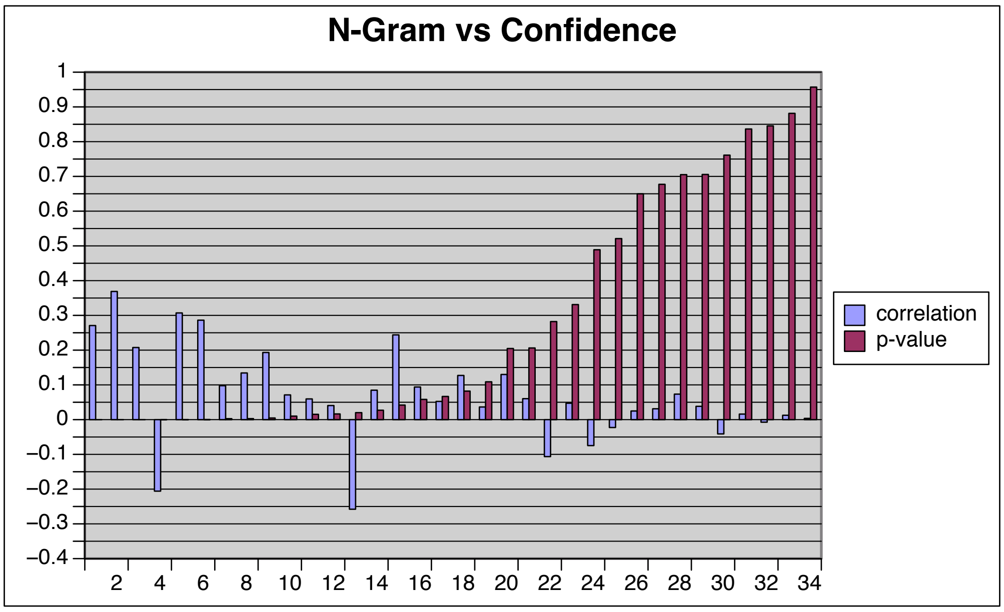


Figure 5.4: Correlation between N-Gram based semantic similarity measures and confidence

Figure 5.4 shows a lack of any substantial evidence to infer that a precise type of correlation (+ve or -ve) exists between semantic and logical coupling strengths. There is a positive correlation in some projects, while a negative correlation in

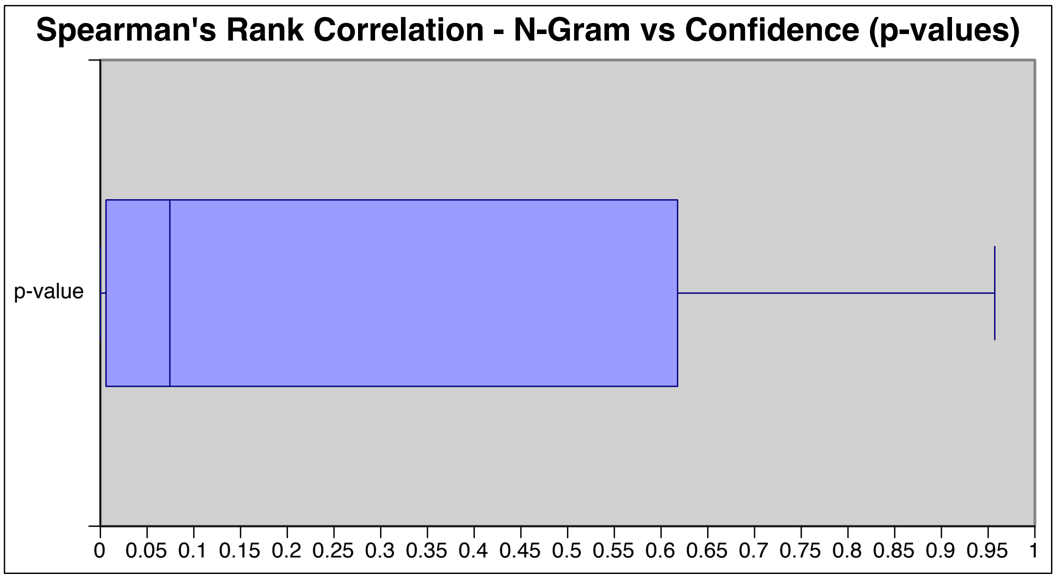


Figure 5.5: Correlation between N-Gram based semantic similarity measures and confidence (box-plot distribution of p-values)

others. The p-values in Figure 5.5 further shows that the correlations might have been identified by chance and are not statistically significant. This is because most of the p-values are above 0.01 except for only a few of the sample of studied projects.

Therefore, due to the lack of any considerable evidence to suggest that there is a linear correlation between semantic and logical coupling strengths or related OO software classes, we fail to reject the null hypothesis (H_0) for RQ3: *There is no linear relationship between the strengths of logical and semantic dependencies.*

In summary, to answer RQ2 we have computed the linear correlation between the strengths of the semantic and logical coupling class pairs, We have used the semantic similarity of class identifiers and the confidence of their co-evolution. The results indicate that these coupling strengths do not co-vary, so that they should be considered independent. A pair of classes with a higher chance to be co-evolved are not necessarily bound to be linked by a semantic link. This overall observation has two effects:

1. inferring the co-evolution degree of class pairs based on the strength of their semantic coupling and *vice versa* will produce many *false positives.*

2. Using semantic coupling information to predict co-evolution will produce a prediction model with *very low precision.*

Previous research by Abdeen *et al.* [1] has shown that combining semantic and structural coupling information when prediction co-change outperforms using either of them individually. However, semantic coupling metrics produced better recall values compared to structural coupling metrics. Research has also shown that the lack of a linear correlation does not imply a lack of causation [153]. As such the absence of a linear correlation between semantic and logical coupling does not infer that the absence of a directional relationship between the coupling types.

The degree of the co-evolution of classes and their semantic coupling are not significantly correlated

5.6 Points of action

In this chapter, results derived from analysing 79 OSS projects show that there is a large overlap between semantic and logical dependencies (see Section 5.5.1). A high number of semantic dependencies include some degree of logical dependency. This observation also applies in the opposite direction when semantic coupling measurement techniques based strictly on English terms are not adopted in the software domain.

Therefore, in this chapter further action is not recommended to maximize the overlap between semantic and logical class dependencies as a majority of the co-change between classes is captured by their implicit semantic dependency. Furthermore, when semantically coupled classes do not require co-changes, it is not helpful to enforce an irrelevant co-change. That will lead to unnecessary maintenance efforts and ripple effects on other classes without the aim of improving software quality.

Notwithstanding, in the previous chapter we observed that not all the structural dependencies can be accounted for by logical coupling and it is not clear whether all the semantic dependencies share a structural relationship. As such, it is imperative to investigate the interplay between semantic and structural dependencies. If there happens to be a minimal overlap between both types of dependencies, then the appropriate point of action will be to maximize their overlap to minimize unnoticeable coupling during change impact analysis.

5.7 Summary of the chapter

In this chapter, we have probed the relationship between semantic and logical class dependencies on the LSe axis with the aim of understanding the implicit causes of co-change, identify any direct influence of semantic coupling on logical coupling and to mitigate unnoticeable class dependencies during change impact analysis.

On the causality of coupling, findings show there is a large overlap between semantic and logical (change) class dependencies. If two classes are semantically coupled, there is a high change that they will co-evolve in the future. However, differently from RQ2 we have shown that the degree of these dependency types do not show a linear correlation. On the strengths of coupling, results showed that there is no linear correlation between the degree or strengths of the semantic similarity between classes and the frequency of their co-change. Statistical results prove that not all highly semantically related classes require frequent co-changes.

The last results on the linear correlation are particularly important: for example, two (semantically similar) class pairs $A \leftrightarrow \hat{A}$ and $B \leftrightarrow \hat{B}$ could share a semantic similarity of 0.7, but not the same degree of co-change: the pair $A \leftrightarrow \hat{A}$ could change much more often than $B \leftrightarrow \hat{B}$. Even so, what this chapter shows is that it is highly likely that the pairs $A \leftrightarrow \hat{A}$ and $B \leftrightarrow \hat{B}$ will co-change at least once or more. In addition, Spearman's rank correlation coefficient only assesses linear relationships but some relationships can be curvillinear [14]. Earlier research has shown that lack of correlation does not imply lack of causation [94, 191, 200].

Following the results presented in this chapter and Chapter 4, it is evident that the ripple effects of changes are propagated along semantic and structural dependencies. In addition, the presence of semantic and structural coupling between classes in OO software projects cannot be inferred based on an observation of logical dependencies.

Therefore, it is more reasonable to investigate the relationship between semantic and structural class dependencies, with the goal of maximizing the overlap between the two dependency types for advancing change impact analysis. The premise is that an understanding of their direct influence on each other and maximizing their overlap (intersections set - classes with both structural and semantic link) will minimize coupling analysis efforts, testing efforts and improve the identification of hidden class dependencies during CIA. Therefore, the interplay between semantic and structural coupling dimensions on the StSe axis is investigated in Chapter 6.

Chapter 6

Interplay on the structural and semantic (StSe) coupling axis

6.1 Introduction

In Chapters 4 and 5, we have empirically investigated the interplay between structural and logical class dependencies, and semantic and logical class dependencies respectively. An important outcome of these chapters is that the strengths of structural or semantic dependencies do not share a linear relationship with the strength of logical dependencies.

The results in Chapters 4 and 5 demonstrate that not all the logical dependencies can be captured by either ONLY structural or semantic dependencies. This is of great significance and has a high impact on dependency based CIA. This is because depending on the type of coupling links between classes under analysis (structural or semantic) during CIA of OO systems, classes with only one type of dependency or link to other classes will be discovered. It is important for developers to be able to precisely trail the path along which the effects of modifications to classes might propagate in a software system. This will reduce future maintenance efforts as well as testing efforts.

In this chapter, we will focus specifically on the interplay between structural and semantic dependencies between OO software classes along the StSe axis with the goal of reducing unnoticeable dependencies during CIA. The rest of this chapter is structured in the following way. In the next section the motivation for this study is presented. For replicability, in Section 6.3, a description of the data collected and the investigated research questions are presented with a worked example in Section 6.4. Section 6.5 outlines the results for the overall sample of OSS projects. Based on these results, refactoring techniques that can maximise the overlap between structural and semantic class couplings are proposed with worked examples in Section 6.6, which is followed by a summary of the chapter in Section 6.7.

6.2 Motivation

Researchers [64, 71, 147, 148] have identified the need to study the interplay between *structural* and *semantic* dependencies, as not all structural or semantic dependencies are related to change dependencies.

Structural and semantic dependencies play a large role in software evolution and change propagation in OO software. It has been shown with a small sample of classes, that structural dependencies exhibit a linear relationship with logical dependencies [203], but it remains unclear whether a stronger structural link implies a strong semantic link. Past research, specifically focused on establishing a link between class dependency types [64, 71, 147, 148] have identified that there is a need to study the interplay between *structural* and *semantic* dependencies, as not all structural dependencies are related to co-evolution. Until now, the interplay between *semantic* and *structural* coupling has not been investigated, despite the fact that an analysis of only semantic dependencies during CIA will not reveal some structural dependencies and vice-versa [1, 102, 156]

According to Yu and Rajlich [205], hidden dependencies make both software comprehension and maintenance hard. They play an important role in software maintenance and evolution because they spread changes among the classes and they can be hard to detect. Kagdi and Maletic have estimated that hidden dependencies are those that cannot be captured by source code analysis alone [100]. Therefore, developers will miss a significant number of them by relying on source code information alone during change impact analysis for example [34].

An example of developers missing a significant number of dependencies is presented the study by Kagdi *et al.* [102] on software coupling based change impact analysis. Therein, they integrated logical and semantic dependency metrics for impact set identification. When comparing the estimated and actual change impact sets, the different dependency metrics estimated different and incomplete sets of classes that might get impacted by given change requests. They also identified that the union of these metrics produced a higher accuracy than their intersection and larger periods of history improve the accuracy of the logical couplings and their combination with semantic coupling. However, there will be cases where historical data is absent. Abdeen *et al.* [1] performed inter-system and intra-system change impact prediction using structural, semantic dependencies and a combination of both, then compared results. Abdeen *et al.* identified that using semantic coupling produces better recall values, in particular, in the intra-system scenario. They mentioned that the inclusion of semantic coupling data provided extra information that deals with the complexity of structural dependencies in the learning phase. On the other hand, they identified that using structural dependencies or a combination of both types of dependencies out-performed the use of semantic dependencies ONLY.

This study takes a different point of view from previous studies in the literature: while previous studies focused on the benefits of combining the two information sources (structural and semantic coupling) [1, 102], this study argues that the need to combine the sources because they are not consistent is an issue in itself as it involves analysing dependencies in multiple ways to achieve a maintenance goal.

Ideally, coupled class pairs should be linked structurally and semantically. If a class makes operational calls to another class, then it expected that the *caller* and *called* classes will both consist of related comments describing the feature being relied upon by the *caller* class. Previous research has demonstrated that this is not the case and forms the core motivation for this study [1, 102, 104, 156].

In contingency Table 6.1, we classified the class dependencies of OO systems into four possible scenarios based on structural and semantic coupling with the use of a 2x2 contingency table. When pairs of classes are linked both structurally and semantically, we posit that their dependency is established ('E') as in Table 6.1. If a structural link is present, but not a semantic one, there could be a strong (denoted 'S') missing dependency. On the contrary, if the semantic link is present, but not the structural one, a weak ('W') missing dependency is detected. When neither a semantic or structural link is detected, no dependency ('x') is established.

The *non-ideal* class pairs are those that lead to unnoticed class dependencies during dependency based CIA, as in cells S and W in Table 6.1.

Table 6.1: Established, hidden and weak dependencies

Structural vs. Semantic Dependencies

	semantic	*not semantic*
structural	E	S
not structural	W	x

The *ideal* class pairs adhere to the following software dependency rule: *for any pair of semantically (symmetrically) coupled classes A, B (A $\leftrightarrow$ B); A and B should at least be structurally (asymmetrically) coupled in any direction (A $\rightarrow$ B or B $\rightarrow$ A) while class pairs* not symmetrically coupled should not be asymmetrically coupled.

An understanding of the interplay between structural and semantic class dependencies has several applications in software engineering and maintenance; (1) it will help increase the precision of estimated impact sets when compared to actual change impact sets, as unnoticeable dependencies based on either a structural or semantic dependency analysis will come to light if the overlap between both dependency types is maximised (2) it will help to point out refactoring opportunities in OO software and finally (3) minimise testing efforts as class with EITHER structural

OR semantic coupling links and not BOTH will be tested after a change request is implemented. It is noteworthy that the actual impact set can vary because changes can be carried out differently [181], thus affecting the precision of estimated impact sets to a certain degree. In this study we will investigate the interplay between structural and semantic class dependencies in our sample of OSS projects.

6.3 Empirical investigation

This study empirically investigates the interplay between structural and semantic coupling, which is Obj3 described in Chapter 1. This section describes the methodology adopted in carrying out this study for replicability on a different sample of OSS projects.

6.3.1 Studied sample of OSS projects

In this study, we have analysed the same sample of 79 OSS projects studied in Chapters 4 and 5. These are all written in Java, of varying sizes and domains, and publicly available as OSS. The meta-data for each project is presented in Table 2 and each project is briefly described in Section A.2.

6.3.2 Data collected

The strengths of the structural and semantic coupling between class pairs are measured by the number of operational calls between them and their semantic similarity respectively. For each of the 79 OSS projects under investigation, we have collected this pair of metrics for all pairs of Java classes following the methodologies outlined in Chapter 3. There is a structural dependency between two classes if the number of operational calls from *caller* to *called* class is greater than zero. There is a semantic link between two classes if their semantic similarity >= 0.5.

6.3.3 The research questions

If the structural and semantic class dependency types are established, the prediction of ripple effects across the system can be run more precisely. On the other hand, when dependencies are unnoticed or weak, developers will detect a smaller number of dependencies capable of propagating further change. Based on these premises, this study aims to answer the following research questions:

- RQ1: Is there a significant association between structural and semantic dependencies in OO software?

  This research question is based on a gap in the literature regarding the absence of an investigation of the interplay between structural and semantic class dependencies in OO software projects. Their interplay has not been studied, even with the fact that an analysis of only semantic dependencies during CIA will not reveal some structural dependencies and *vice versa* [1, 102, 156]

  The premise is that overall there will be a small overlap between structural and semantic class dependencies in the studied sample of OSS projects.

- RQ2: Is there a significant linear relationship between the strengths of structural and semantic coupling? Similarly to RQ1, it remains unclear whether one can infer a strong semantic similarity between classes based on an observation of a high number of operational calls from a *caller* to a *called* class.

6.4 Worked example

In Chapter 3, we defined structural and semantic coupling as well as their extraction of related metrics. In the same chapter we also identified the efficiency gained by measuring the semantic coupling between classes based on their identifiers. We further identified that the N-Gram sentence similarity technique performs better on English and non-English terms compared to information retrieval techniques for sentence similarity which rely on the English language dictionary for the meaning of words. Thus, semantic coupling metrics in this study are based on the N-Gram sentence similarity technique.

In this section, we demonstrate the study methodology with a project from our sample as a worked example for replicability. The project under analysis is the *monome-pages*[1] project (project ID = 142 in Table 2). According to the project owners, “Pages is a monome application that allows the simultaneous execution of multiple other monome applications on any number of devices. There are many built in applications as well, including MIDI interfaces (keyboard, triggers, faders, sequencers), Ableton clip launcher interfaces, and a way to easily script your own programs with Groovy”. As at the time of the data extraction for this study, the project had undergone 256 revisions and composed of 835 structural class dependencies, 1,429 semantic class dependencies.

6.4.1 RQ1: Overlap between structural and semantic coupling

By answering RQ1, we test for the following null hypothesis H_0: *Structural and semantic class dependencies are independent.*

In order to answer RQ1, it is imperative to identify the number of distinct structural and semantic dependencies in each project. Afterwards, the contingency table is populated as in Table 6.1. Table 6.2 illustrates the populated contingency table for the example project - *monome-pages.* This project has a total of:

- 167 class pairs with both structural and semantic links in cell E;
- 582 class pairs with ONLY structural links in cell S;
- 555 class pairs with ONLY semantic links in cell W, and;
- 12 class pairs without any links in cell x.

This observation is further presented in Figure 6.1.

[1] code.google.com/p/monome-pages

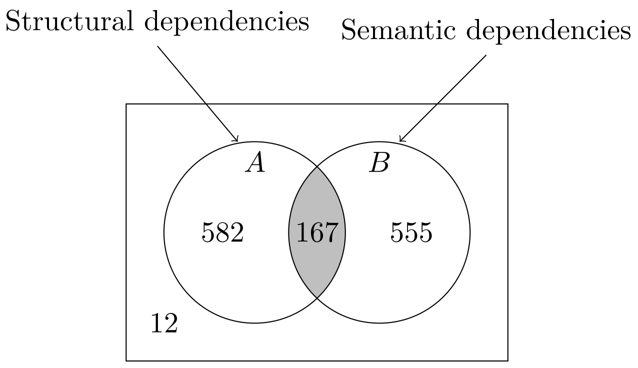


Figure 6.1: Intersection of structurally and semantically coupled classes in *monome-pages*

Table 6.2: Established, hidden and weak dependencies

Structural vs. Semantic Dependencies in Monome-Pages

	semantic	*not semantic*
structural	167	582
not structural	555	12

We take two examples of class pairs in cells W and S of Table 6.2 from the latest snapshot of the source code for further analysis.

As an example of class pairs in cell W, we consider the semantic dependency `AbletonClipDelay.java` ↔ `AbletonClipLauncherPage.java`. This pair of `.java` classes do not have any operational calls between them or source code links. However, conceptually, their identifiers are related because of the presence of similar terms upon splitting the identifiers (i.e., "Ableton", and "Clip"). In line with this, their semantic similarity metric as derived from the N-Gram technique is 0.6, above the dissimilarity threshold of 0.5 and these classes have been co-changed in four revisions. This demonstrates that ripple effects are not only propagated *via* structural couplings or source code links. Semantic dependencies are also related to co-evolution, as demonstrated by empirical results in Chapter 5.

A closer look into the source code of these classes, reveals that they also share similar terms in their comments. Listings 6.1 and 6.2 illustrate excerpts from the source code of the classes texttttAbletonClipDelay.java and `AbletonClipLauncherPage.java`, respectively. The listings show that the source code and comments of

```
 1  package org.monome.pages.ableton;
 2
 3  import org.monome.pages.Main;
 4  import org.monome.pages.configuration.Configuration;
 5
 6  /**
 7   * Delays sending a play clip command to Ableton, used by the Live Looper page to cut
        loops.
 8   *
 9   * @author Tom Dinchak
10   *
11   */
12  public class AbletonClipDelay implements Runnable {
13
14          /**
15           * Amount of time to delay in ms.
16           */
17          private int delay;
18
19          /**
20           * The track number where the clip lives, from left to right starting at 0.
21           */
22          private int track;
23
24          /**
25           * The clip number, from top to bottom starting at 0.
26           */
27          private int clip;
28  ...
```

Code 6.1: AbletonClipDelay.java

both classes contain a high occurrence of the term “clip” embedded within them. This is visible in lines 7, 20, and 25 of Listing 6.1 and lines 3, 6, 8, 12, 13 of Listing 6.2. There is no source code link between these classes.

As an example of class pairs in cell S of Table 6.2 we shall consider the structural dependency `Configuration.java` → `AbletonClipDelay.java`. The *caller* class here is `AbletonClipDelay.java` with three operational references to `Configuration.java`. These three source code links can be seen in lines 2, 5, and 18 of Listing 6.3. Excerpts of `Configuration.java` are also shown in Listing 6.4. It is immediately clear from Listing 6.3 why there is no semantic link between both classes. The lines with operational calls to `Configuration.java` do not have any comments within the source code in Listing 6.3 to explain them. Comments hold valuable information that can

```
...
        /**
         * Sends "/live/play/clip track clip" to LiveOSC.
         *
         * @param track The track number to play (0 = first track)
         * @param clip The clip number to play (0 = first clip)
         */
        public void playClip(int track, int clip) {
                this.monome.configuration.getAbletonControl().playClip(track, clip);
        }

        public void stopClip(int track, int clip) {
                this.monome.configuration.getAbletonControl().stopClip(track, clip);
        }

        /**
         * Sends "/live/arm track" to LiveOSC.
         *
         * @param track The track number to arm (0 = first track)
         */
        public void armTrack(int track) {
                this.monome.configuration.getAbletonControl().armTrack(track);
...
```

Code 6.2: AbletonClipLauncherPage.java

help developers in understanding the source code and is leveraged when measuring semantic coupling and cohesion [156]. It has been recommended that there should always be sufficient comments or documentation within source code. This will describe what has been implemented[2]. This will further introduce a semantic link between the classes in Listings 6.3 and 6.4. As a result they will belong to set of established class dependencies in cell E of Table 6.2.

After generating the 2x2 contingency table for the project, we further carry out independence tests to investigate the presence of a significant association between structural and semantic class dependencies. In this study we have adopted the Fisher's exact independence test. It asserts the independence of two categorical variables. In this study, the variables under analysis are structural and semantic class dependencies. This test is adopted because it performs better than the Chi-square X^2 test when some cells in the contingency tables have values of zero (0). If

[2]Comments within source code improve program comprehension for software developers and maintainers [124]

```java
...
        private Configuration configuration;
...
        public AbletonClipDelay(int delay, int track, int clip, Configuration configuration)
            {
                this.configuration = configuration;
                this.delay = delay;
                this.track = track;
                this.clip = clip;
        }

        public void run() {
                try {
                        Thread.sleep(this.delay);
                } catch (InterruptedException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                }
                this.configuration.getAbletonControl().playClip(track, clip);
        }
...
```

Code 6.3: AbletonClipDelay.java

the test returns a p-value of > 0.01, we reject the null hypothesis associated with RQ1. Similarly to the empirical study in Chapter 4, in this study we reject the null hypothesis at the 99% confidence level to minimise the chance of a type I error – mistakenly rejecting a null hypothesis.

Based on the data in Table 6.2, the Fisher's exact test of independence results returned a p-value of less than 0.01 (4.8e-197). A relatively similar value is also returned by the Chi-square statistical test (2.8e-1). As a result of the outcome of the Fisher's exact test for independence, we reject the null hypothesis and fail to reject the alternative hypothesis H_1: *Structural and semantic class dependencies are significantly associated.*

From these statistical results, we can infer that when there is a structural link between classes we should expect a semantic link but no semantic links if there is no structural link. In other words, there is a significant association between structural and semantic class dependency. However, looking at Table 6.2 it is evident that there is a higher number of coupled classes in cells S and W compared to cell E. This

```java
  ...
/**
 * This object stores all configuration about all current monomes and pages. It
 * also stores global options like Ableton OSC port selection and enabled MIDI
 * devices.
 *
 * @author Tom Dinchak
 *
 */
public class Configuration implements Receiver {

        /**
         * The name of the configuration.
         */
        private String name;

        /**
         * The number of monomes currently configured.
         */
        private int numMonomeConfigurations = 0;

        /**
         * An array containing the MonomeConfiguration objects.
         */
        private ArrayList<MonomeConfiguration> monomeConfigurations = new
             ArrayList<MonomeConfiguration>();

        /**
        ...
```

Code 6.4: Configuration.java

observation calls for refactoring techniques to minimise the number of links in cells S and W and maximise the number of links in cell E to reduce unnoticeable class dependencies during change impact analysis based on either structural or semantic coupling information.

6.4.2 RQ2: Linear relationship between structural and semantic coupling

In resolving RQ2, we test for the following null hypothesis H_0: *There is no linear relationship between the strengths of structural and semantic class dependencies.* In this Thesis, we have tested for linear relationships by means of the Spearman's rank correlation test as described in Chapter 3.

To conduct the Spearman's rank correlation test, we have generated two vectors based on the strengths of structural and semantic class dependencies in cell E of the contingency Table 6.2. The rationale behind this is that we want to probe whether the number of structural and operational links between classes has an effect on the degree of their semantic similarity and *vice versa.* One vector represents the number of operational calls between the classes while the second represents the semantic similarity metric of the classes. This test will determine whether these two metrics co-vary; whether a high semantic similarity means a high number of structural links between the classes and *vice versa.*

Spearman's rank correlation output for the example project – *monome-pages* reveals a low positive correlation coefficient of 0.1. However, the p-value is 0.07 and insignificant (> 0.01) considering our chosen α=0.01. Based on these results, we cannot reject the null hypothesis for RQ2 for *monome-pages.*

6.5 Results: Overall sample of OSS projects

In this section we report the results derived from the analysis of the overall sample of OSS projects following the methodology in Section 6.4 in answering RQ1 and

RQ2 and testing for their associated null hypotheses.

6.5.1 RQ1: Overlap between structural and semantic coupling

To recap, RQ1 posed the following question: *Is there a significant association between structural and semantic dependencies in OO software?* To answer RQ1, firstly we populate the contingency table (i.e., Table 6.1) *per* project. Table 8 presents the data for each cells of the contingency table *per* project and p-values derived from the Fisher's exact independence tests. An excerpt of this table is presented in Table 6.3.

The first column shows the project names, the next four columns show the number of established dependencies (E), strong hidden dependencies (S), weak hidden dependencies (W) and class pairs without any dependencies (x). The sixth column shows the p-values derived from the Fisher's exact Independence test *per* project while the seventh column shows the p-values derived from the Chi-square statistical test.

As shown in Table 6.3, three projects have no class pairs in cell x. This is because all the classes in those projects either have an outgoing or incoming source code link to one or more classes; a semantic similarity above the threshold of 0.5 between the identifiers of each class to at least one other class in the project or both a structural and semantic link to on or more classes.

Overall, the p-values returned by the Fisher's exact independence test indicate that structural and semantic coupling share a significant relationship. A majority of projects showed a significant association between structural and semantic class coupling. We further derived similar results (p-values < 0.01) for a significant majority of the projects using the Chi-square test for independence to confirm initial Fisher's exact test results and check whether the low p-values derived from the Fisher's test are due to overflow errors (the influence of larger values in some cells). Chi-square independence test results for all the OSS projects in the studied sample are presented in Table 9. Therefore, based on statistical results, we can reject the

Table 6.3: E, S, W and x sets of class pairs (excerpt from 20 studied OO software projects)

Project Name	E	S	W	x	p (Fisher's)	p (Chi-square)
2dtetris	16	64	5	0	0.0006	0.0005
4-connect	0	18	0	1	1	Nan
ahs-scheduling	6	52	1	3	0.4	>0.9
aima-java	547	2,875	2,627	193	0	0
alexo-chess	111	527	266	1	0	3.7e-115
algmusic	37	179	166	29	0	1.5e-42
alleywayreinvented	10	101	12	16	0.0001	4.2e-05
alto	258	1,328	1,809	28	0	0
amock	119	931	72	2	0	0
apjava	8	35	8	4	0.003	0.004
appletbomberman	40	222	77	1	0	1.98e-41
ascrblr	33	126	59	6	0	1.8e-21
audao	88	220	832	38	0	3.1e-131
bitlyj	45	141	76	3	0	2.2e-26
bluecove	142	559	1,338	423	0	0
castanea	18	122	22	6	0	5.6e-13
catchnthrow	5	42	10	0	0	5.6e-08
daedalum	35	211	98	78	0	4.2e-19
dbmigrate	9	4	2	0	1	1
echo-nest-java-api	10	103	15	21	0	1.5e-05

null hypothesis for the overall sample and fail to reject the alternative hypothesis H_1: *Structural and semantic class dependencies are significantly associated.*

Table 6.4 outlines descriptive statistics for the p-values derived from Fisher's exact independence test. This table further shows that on average, we stand a 5% chance of wrongly rejecting the null hypothesis. We observed p-values of 1 when more than one cell of the contingency table is empty. For example in the *project-armageddon* OSS project with blank W and x cells, and *4-connect* OSS project with blank E and W cells.

Looking at Figure 6.2 which summarizes the proportion of links belonging to each contingency table cell in the overall sample, it becomes obvious that cells S and W have a larger proportion of class dependencies. It is imperative to rework these class coupling links to facilitate the reduction of unnoticeable class dependencies during dependency based change impact analysis. This is because (as shown in Figure 6.2) only 10% of the links in our sample on average will be noticeable by

Table 6.4: Descriptive statistics of Fisher's exact independence test p-values

Metric	Value
Mean	0.05
Median	0
Mode	0
Skewness	4.4
Range	1
Minimum	0
Maximum	1
Count	79

both structural and semantic coupling analysis. Around 58% of the coupling links will be noticed by an analysis of only structural coupling. This is a high proportion of unnoticed dependencies and propagators of ripple effects during CIA. Finally, around 30% of the links will be noticed by semantic analysis alone.

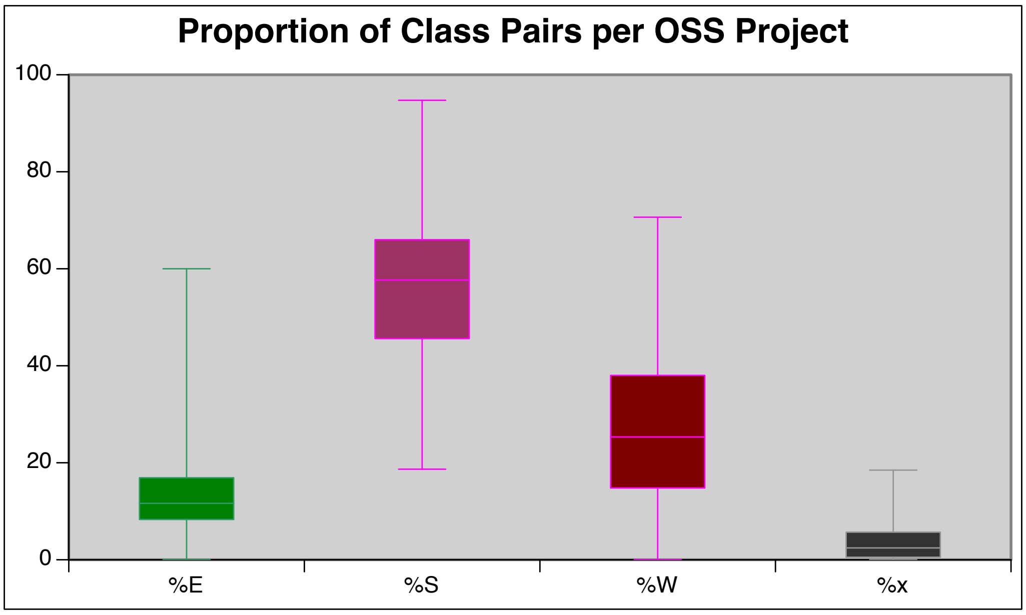


Figure 6.2: Proportion of class pairs belonging to each cell in contingency Table 6.1 *per* OSS project in studied sample

Statistical results provide evidence that structural and semantic class dependencies are significantly associated. The S and W sets of class pairs in Table 6.3 demonstrate the importance of refactoring. For the purpose of reducing unnoticeable coupling during change impact analysis.

6.5.2 RQ2: Linear relationship between structural and semantic coupling

In this section, we report Spearman's rank correlation outcomes for the overall sample of OSS projects. The rationale behind RQ2 is investigating whether the structural and semantic class coupling strengths co-vary or have a significant impact on each other. For instance, we want to detect whether a high number of structural links between classes means a high semantic coupling between the classes.

Overall results are presented in Table 10. An excerpt of that table with correlation results of 20 OSS projects from the studied sample is shown in Table 6.5. The first column shows the project names, the second column shows the correlation coefficient *per* project, while the third and last column shows the derived p-value (measure of significance). A p-value of less than or equal to 0.01 indicates a significant correlation between structural and semantic class dependency strengths (with 99% confidence). Based on the values in Table 6.5, a majority of the projects in our sample do not exhibit a significant correlation. This result is echoed by the box-plots in Figure 6.3, the median p-value therein is around 0.4.

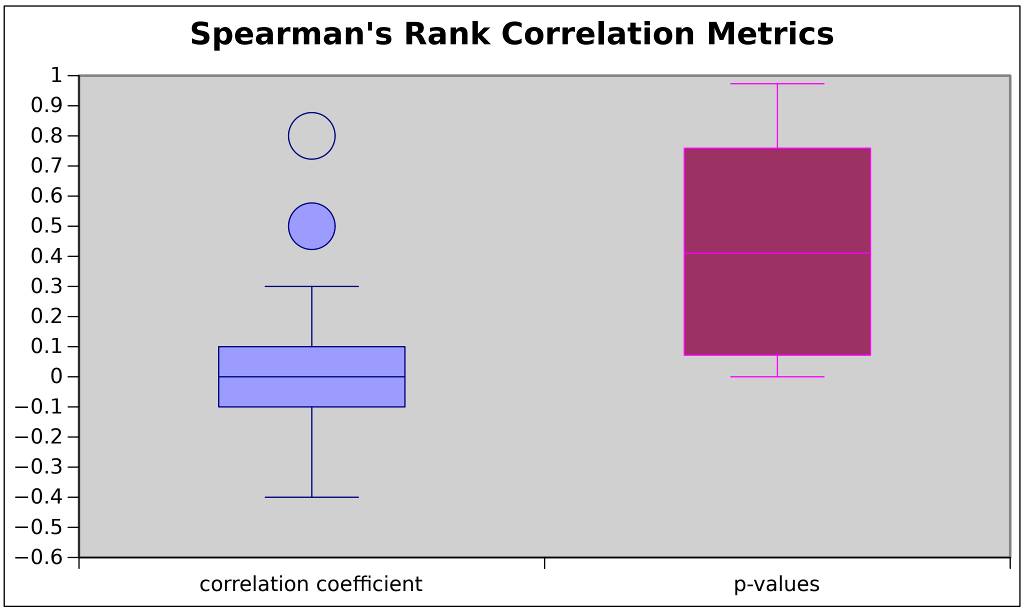


Figure 6.3: Structural and semantic coupling - Spearman's rank correlation results with p-values

Table 6.5: Spearman's rank correlation (excerpt from 20 OSS projects))

Project Name	Correlation Coefficient	p-value
2dtetris	0.1	0.6
aima-java	0.2	< 0.001
alexo-chess	-0.4	< 0.001
algmusic	-0.2	0.4
alleywayreinvented	-0.1	0.7
alto	0.1	0.1
amock	0.0	0.9
apjava	-0.4	0.3
appletbomberman	0.0	0.8
ascrblr	0.0	0.9
audao	0.0	0.8
bitlyj	-0.3	0.1
bluecove	0.1	0.4
castanea	-0.2	0.3
daedalum	0.1	0.6
dbmigrate	-0.1	0.9
echo-nest-java-api	-0.3	0.4
fyllgen	-0.1	0.5
geocoder-java	-0.2	0.2
google-voice-java	0.3	0.1

Highlighted in Table 6.5 are the results with significant p-values (<= 0.01). Two projects exhibit a significant but low to moderate correlation. The *aima-java* project exhibits a significant low positive correlation coefficient of 0.2 with a p-value of 0.00001, while *alexo-chess* exhibits a significant moderate negative correlation coefficient of -0.4 and a p-value of 0.00003. A negative correlation between two variables means that as one variable increases, the other variable decreases. A deeper look into the structural and semantic coupling strengths between class pairs in *alexo-chess*, reveals that the higher the number of operational calls between the `.java` classes, the lower the semantic similarity of the classes. This is illustrated with Figure 6.4

In Table 10, there are two outlier projects with positive correlation coefficients higher than 0.3. The first one is the *jmemcache* project with a correlation coefficient of 0.5, however the p-value is 0.3. The second is the *robost-coupe* project with a significant and large positive correlation coefficient of 0.8 and a p-value of 0.

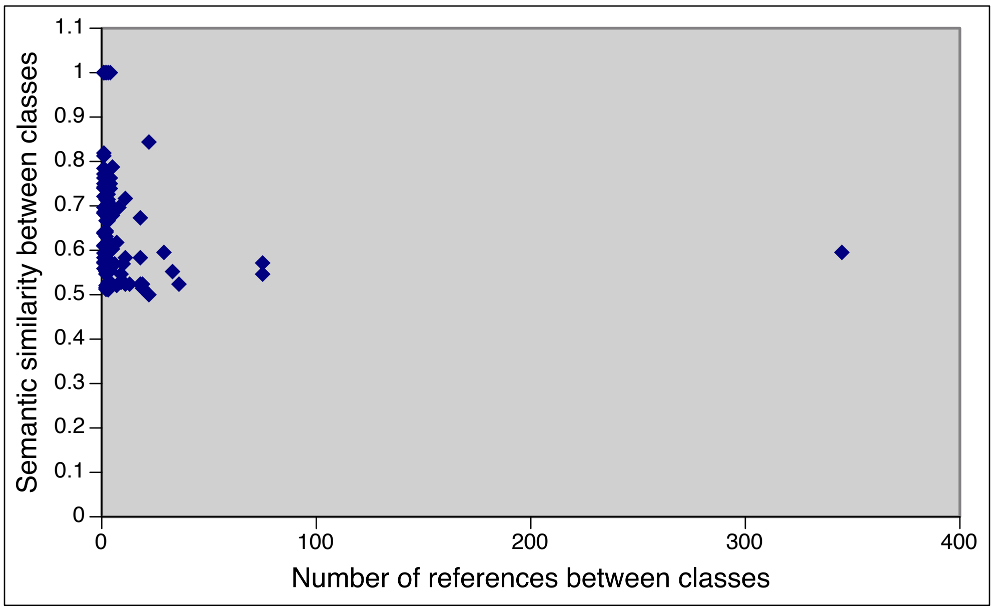


Figure 6.4: Structural vs. semantic coupling strengths in the *alexo-chess* OSS project

However, based on the overall results we cannot infer that there is a significant correlation in OSS projects given the absence of any significant evidence in a majority of the projects to support this claim. Therefore, we have failed to reject the null hypothesis for RQ2 H_0: *There is no linear relationship between the strengths of structural and semantic class dependencies.*

Structural and semantic coupling strengths have minimal effects on each other and do not significantly co-vary. A strong structural link cannot be inferred from a strong semantic link

6.6 Points of action

Results derived from analysing 79 OSS projects and presented in Section 6.5.1 have shown that there is a small overlap between structural and semantic class dependencies but a significant association between these two coupling types.

A majority of projects exhibit a larger number of class pairs in cells S and W of the motivating contingency Table 6.1. Having identified that it is imperative to decrease unnoticeable coupling during CIA and ss a means of resolving RQ3 presented in Section 6.3.3, we are proposing two practical techniques to maximize the overlap between structural and semantic class dependencies in OO software.

Mitigating unnoticeable class coupling during CIA has several applications in software engineering and maintenance as outlined in Section 6.5.1. The two techniques are presented in Section 6.6.1 and 6.6.2 and are geared towards:

- Resolving the S dependencies, and
- Resolving the W dependencies

6.6.1 Solving the S dependencies: 'rename class' design pattern

In Section 6.4.1, we presented an example of class pairs belonging to the S cell (i.e., the hidden dependencies). These classes are structurally linked but not semantically linked. An analysis of semantic coupling during change impact analysis will not reveal these classes to be coupled because of the absence of similar terms and phrases in their identifiers or source code.

Prior studies have laid a great emphasis on the benefits of meaningful identifiers and comments within source code because of the vital information they provide [129] when analysing coupling and co-change. Ujhazi *et al.* propose the importance of improving the quality of the underlying textual information in source code because semantic coupling metrics rely on this information. This can be done by *"applying advanced source code pre-processing techniques for splitting and expanding identifiers and comments in software"* [187].

Both Fisher's and Chi-square statistical independence test results in Section 6.5.1 have shown a significant association between structural and semantic class coupling. Therefore, with the goal of minimizing the number of hidden dependencies during CIA and strengthening this association, we propose two refactoring activities for OO software class pairs belonging to the S cell:

- the addition of meaningful comments in both the caller and called classes as well as the affected methods;
- the use of conceptually related identifiers by means of renaming related methods and classes, using the "Rename Class" design pattern [68].

For example, operations of classes referenced by other classes should be described with proper comments in both the class being referenced and the class making the reference. Analysis of the information embedded in identifiers and comments is useful for a number of software development and evolution tasks [10, 42, 124].

6.6.2 Solving the W dependencies: a novel semantic and structural link 'extract class' design pattern

In this section, we propose a new 'extract class' refactoring pattern for resolving the W dependencies in OO software.

Design patterns provide a common vocabulary for designers to communicate, document and explore design alternatives. A good set of design patterns effectively raises the level at which one programs [69]. They can be considered microarchitectures that contribute to the overall software architecture.

Gamma *et al.* [69] defined design patterns, their classification, a means to describe them using a design pattern template and a catalog of design patterns they had discovered. In this section, we have adopted the design pattern description template in describing our proposed pattern for resolving the W set of class dependencies in OO software. Our proposed semantic and structural link design pattern is described as follows:

Intent. *The intent of a pattern answers several questions including: What does the design pattern do? What is its intent and rationale? What design problem does it solve?*

The proposed pattern serves to maximize the overlap between structural and semantic class dependencies in OO software where there are semantic but no structural links between classes. A lack of an overlap will increase maintenance efforts when attempting to carry out impact analysis or proposed changes.

Motivation. The motivation of a pattern is concerned with a scenario in which the design pattern is applicable, the particular design problem or issue the design pattern addresses and the class and object structures that address the issue [69].

Results in Section 6.5.1, showed that the projects in our sample have a significant number of class pairs in cell W of contingency Table 6.1. Differently from class pairs that belong to cell S, these are linked conceptually but not structurally. Among the examples presented in Section 6.4.1, the class pair `AbletonClipDelay.java` and `AbletonClipLauncherPage.java` have strongly related identifiers, as well as words in the comments embedded in their source code.

The research domain that can facilitate the refactoring of the class pairs in the W subset is referred to as *Extract Class* (EC) refactoring [19]. EC is a refactoring approach that analyzes the (structural and/or semantic) similarity of the methods in a class in order to identify chains of strongly related methods (i.e., *method chains*). Those identified method chains are further adapted to define new classes with higher cohesion than the original class. Using a shared definition, cohesion is the *"degree to which elements of a module belong together"* [30] and classes are a set of responsibilities [129]. Past findings in this context states that *"classes with unrelated methods often need to be restructured by distributing some of their responsibilities to new classes, thus reducing their complexity and improving their cohesion"* [19].

Research has shown that the EC approach is able to identify meaningful refactoring operations and new cohesive classes. Bavota *et al.* [19] improved on their previous work [20] by proposing a better extract class refactoring approach beginning with one class to automate extract class refactoring with the goal of improving the cohesion of classes in OO software. Their approach combines structural and semantic coupling and cohesion information.

With lessons learned from earlier research, we are proposing an improved EC approach based on the semantic similarity of methods belonging to class pairs in cell W (Table 6.1). This is geared towards minimizing the variance in the class dependencies detected by structural and semantic coupling analysis during CIA. Differently from the previous approach proposed by Bavota *et al.*, our approach also extracts chains of related methods from the class pairs, but based on the semantic similarity of the methods. It is noteworthy that this is ONLY done when the classes do not have a strong internal structure based on semantic cohesion [43, 129], which is also an important attribute of OO software classes. According to Kabaili *et al.* [97] *"some classes have multiple methods that share no variables but perform related functionalities and putting each method in a different class would be against good OO design".*

Applicability. The applicability of a pattern answers several questions such as what are the situations in which the design pattern will be applied? what are examples of poor design that the pattern can address? [69].

Enabling conditions for using the proposed pattern are:

- the structural and semantic dependency pattern of an OO software must be analysed.
- there must be semantically related and non-structurally related classes in Cell W as illustrated in Table 6.1.

Based on these enabling conditions, our proposed pattern is applicable:

- when there are classes with semantic links but no structural links in and OO software.

To become useful, a design pattern should be applicable in multiple problem domains [69]. Software maintenance domains where our proposed pattern is applicable are the CIA, software test emulation and fault prediction domains given a wide overlap between semantic and structural class links.

Participants. Patterns vary in their level of granularity and level of abstraction and are thought of in terms of two orthogonal criteria. Jurisdiction which is the level of abstraction (class, object, or compound) and characterization (i.e., what a pattern does: structural, creational or behavioral).

Our proposed semantic and structural link extract class design pattern spans the class jurisdiction because it deals with the relationships between classes and their composition (methods). Its characteristic spans structural because it is concerned with the composition of classes [69].

Participants are the classes and objects participating in the design pattern and their responsibilities [69]. The participants in our proposed pattern are the target pair of classes (semantically related and non-structurally related classes in Cell W Table 6.1) from which method chains are extracted which lead to new or modified classes.

Diagram. Design patterns provide a target for the reorganization or restructuring of software artifacts at various levels of granularity (e.g., classes and methods) [65][69]. As such, design patterns are automatable as refactorings [184]. For example, the singleton[3] refactoring converts an empty class into a singleton.

Figure 6.5 presents a pictorial view of the refactoring automation of our proposed pattern in the form of an extract class refactoring workflow.

Consequences and Implementation. In Figure 6.5 we propose that the newly created classes are a combination of already existing structurally coupled, classes and their closest semantically coupled extracted method chain from only classes with weak internal structures. This design decision was taken in order to not increase the overall coupling, and to preserve the software

[3]The singleton pattern is a design pattern in software engineering that restricts the instantiation of a class to only one object.

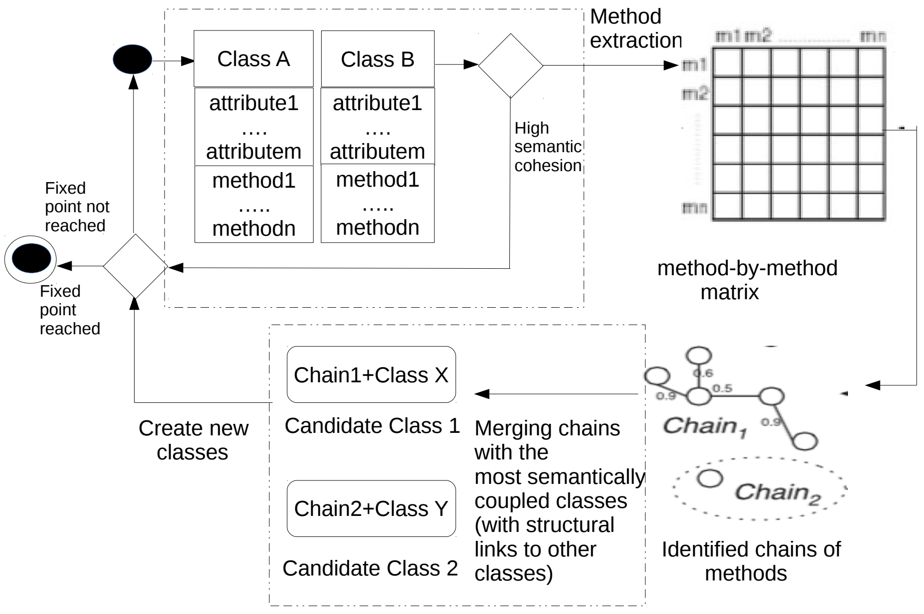


Figure 6.5: Extract class refactoring for pairs of semantically related and non-structurally related classes in Cell W Table 6.1), partly adapted from [19] Fig. 1. (dashed components are our novel additions to the original model)

architecture, while increasing the semantic cohesion of classes. If new classes are created only using the extracted chain of semantically related methods, the hidden dependencies would remain undetected during CIA: those classes would be semantically cohesive, but without structural links to other classes. This is one of the pitfalls to be aware of when implementing this pattern.

To avoid having new classes with a very low number of methods, Bavota *et al.* [19] merged each trivial method chain with the most coupled non-trivial method chain to obtain the final set of classes to be extracted from the original class. In our adapted approach, each method chain is merged with the most semantically coupled (already existing) class, which in turn must have structural links to one or more classes.

6.7 Summary of the chapter

In this chapter, we have probed the association and overlap between structural and semantic coupling. While previous studies have focused on combining structural and semantic coupling information, we have argued that the inconsistency in their information is a gap in itself due to the inefficiency in measuring coupling in multiple ways to achieve one maintenance goal. Firstly, we statistically investigated whether a significant association exists between structural and semantic class dependencies.

Although Fisher's as well as Chi-square independence test results showed that there is a significant association between the coupling types; for some projects the values in cell S are larger than those of cell E of Table 6.1. Therefore, the conclusion based on results of RQ1 is that usually classes associated by a structural link are not semantically linked. This means that a small overlap between semantic and structural class coupling has been observed in our studied sample of OSS projects. In addition, carrying out dependency based change impact analysis based on either structural or semantic coupling information in our studied sample of projects will lead to unnoticeable coupling among classes.

Secondly, we have investigated the presence of a linear relationship between structural and semantic class dependencies in all the 79 OSS projects in our sample. This is to find out whether a high number of operational calls between classes translates to a high degree of semantic similarity between the classes. Interestingly, we revealed by means of the Spearman's rank correlation that this is not the case. A strong semantic similarity cannot be inferred from an observation of a structural link between any class pair.

Finally, having identified a small overlap between structural and semantic class dependencies, we presented two techniques in Sections 6.6.1 and 6.6.2 that can help to maximize the overlap between structural and semantic class dependencies, minimize testing efforts as well as minimize unnoticeable class coupling during dependency based change impact analysis in OSS projects at the class level of granularity. Chapter 7 consists of a summary of the Thesis and planned actions for further research.

Chapter 7

Conclusion

7.1 Introduction

The Thesis is composed of a three-fold empirical study on the influence of three software dependency types on one another. This chapter synthesizes the highlights of this Thesis. Firstly, the relationship between structural and logical dependencies was investigated. This was immediately followed by an investigation of the interplay between semantic and logical dependencies. Lastly, the interplay between semantic and structural class dependencies was investigated in the context of `.java` classes in OO software.

This Chapter begins with an overall summary of the empirical studies conducted in Section 7.2; including the main contributions in Section 7.3, their beneficiaries and impact on software engineering research and practice. In Section 7.4, achievements derived from the research experience are summarized. After that, the research limitations are discussed and with feasible directions for future research in Sections 7.5 and 7.6, respectively.

7.2 Research summary

The main research objectives of this Thesis as summarized in Section 1.6 of Chapter 1, are to investigate the interplay between:

1. Structural and logical software dependencies
2. Logical and semantic software dependencies
3. Structural and semantic software dependencies

In order to address the aforementioned research objectives, we carried out a review of the literature as described in Chapter 2 to identify existing findings and gaps in the body of existing knowledge. After this, we described the methodology of the research in Chapter 3, including the sampling of the studied sample of OO software projects, statistical tests and software metrics applied in this research.

The initial empirical study was investigated and described in Chapter 4. This study was carried out to probe the relationship between the structural and logical coupling of classes in OO software. The results of this study demonstrated that the majority of the class pairs with a structural link are expected to undergo co-changes. However, not all the logical class dependencies are structurally linked. This investigation also showed that the degree of the structural relationship between classes has no effect on the degree of their co-evolution frequency. Differently from prior empirical studies, that tend to emphasise on the 80-20 Pareto distribution[1] in most of the software metrics on single software artifacts (complexity [45], defect density [60], number of changes [112]), the results from Chapter 4 have provided statistical evidence that structural and logical class dependencies in OO software projects do not follow such distributions.

Majority of the class pairs linked structurally are expected to undergo co-changes but not all co-changes classes share a structural link.

[1]the law of the vital few states that 80% of the effects come from 20% of the causes

In Chapter 5, the interplay between semantic and logical class dependencies was investigated. An investigation of whether the semantic similarities of class pairs can be measured by their identifiers alone in comparison to analysing the terms embedded internally was also investigated in this chapter. The results from Chapter 5 have shown that measuring semantic coupling at the class level of granularity is more efficient just by analysing class identifiers split in form of short sentences. Similarly to Chapter 4, the results of this study also demonstrated that majority of the semantic class dependencies would normally undergo co-changes. However, not all the logical class dependencies are conceptually linked. This investigation also showed that the degree of the semantic relationship between classes has no effect on the degree of their co-evolution. In addition, the semantic and logical class dependencies in OO software projects do not adhere to the Pareto's rule of the vital few.

Semantically related classes would normally undergo co-changes but the degree of the semantic relationship between classes has no effect on the degree of their co-evolution.

With the OO software class dependency insights gained from Chapters 4 and 5 on the interplay between structural and semantic class dependencies and co-evolution, in Chapter 6 we set out to understand the link between structural and semantic class dependencies. In contrast to previous studies that have combined these two sources of information when carrying out change impact analysis (CIA), we identified the inefficiency in this combination as it involves duplicated efforts in measuring coupling. The idea was to investigate the relationship between structural and semantic class coupling and maximize their overlap to minimize CIA efforts. Results showed that their overlap was small and a high structural dependency cannot be inferred based on an observation of a high semantic dependency between OO software classes.

There is a small overlap between structural and semantic coupling.

As a result, we proposed a pair of refactoring techniques to maximize the overlap in two scenarios. In the first scenario, classes are linked semantically but not structurally. In this scenario, we proposed an extract class refactoring approach. In the second scenario, class pairs are linked structurally but not semantically. In this scenario, we have proposed the use of the class renaming design pattern as well as the inclusion of adequate comments embedded in the source code of classes to describe the structural link and features shared between them.

7.3 Research contributions and beneficiaries

The contributions of this thesis as outlined in Chapter 1 are six-fold, and can be presented as follows:

C1 – An interplay between software dependencies. This Thesis has probed and synthesized the interplay between three types of software dependencies. Namely, structural, logical and semantic coupling. This contribution is novel and has added knowledge to the state of the art in the software engineering literature. Both a linear and directional relationship was explored among the software coupling types and a highlight of the findings is the existence of evolutionary consequences of structural and semantic coupling in a significant sample of OSS software projects.

C2 – The role of hidden dependencies. This Thesis has contributed to the body of knowledge on software dependencies by revealing the proportion of hidden dependencies in OO software based on an analysis of anyone of the three types of static coupling (logical, structural and semantic) investigated.

Our empirical studies have shown that only 10% of the couplings in our sample are noticeable by both structural and semantic coupling analysis; 58% of the couplings are be noticed by an analysis of only structural coupling, and around 30% of the couplings are noticed by semantic analysis alone. This is a high proportion of unnoticed dependencies and propagators of ripple effects.

C3 – Refactoring the hidden dependencies. We have proposed novel refactoring approaches based on structural and semantic coupling metrics by which the number of hidden dependencies can be minimized during impact analysis at the class level of granularity depending on the type of coupling analyzed. This is to better improve software maintenance and assist practitioners in minimizing ripple effects of software changes more efficiently. If the overlap between structural and semantic dependencies is large, then these tests only have to be carried out once. Either ONLY semantically coupled class pairs or ONLY structurally coupled pairs will need to be tested. The proposed approaches can inform the implementation of tools such as integrated development environment (IDE) plug-ins to support software development and maintenance and to reduce unnoticeable coupling during change impact analysis and impact set prediction.

C4 – Prediction of software changes. Geipel and Schweitzer [71] state that the question about the causes of change propagation has been overlooked by many researchers in favour of a predictive approach. As such, these causes are implicitly contained in a prediction function or as input to a machine learning algorithm [89, 126, 185, 202, 210]. A strong relationship between logical and structural as well as semantic coupling provides statistical support for these models and predictions, thus helping to achieve more focused software maintenance.

Geipel and Schweitzer rightly state that any model that tries to infer structural coupling from co-change data will produce a lot of false positives [71] because the proportion of change dependencies are always larger than the proportion of structural dependencies [147, 148]. On the other hand, using the structural coupling information between pairs of classes to predict unplanned future co-changes is a more realistic objective as identified by Oliva and Gerosa [147]. But we suggest that this is a realistic objective with the support of semantic coupling metrics.

The reason is that the overlap between structural and logical dependencies is not large. Also the overlap between semantic and logical coupling is not large. *More importantly, we have identified that structural and semantic dependencies do not account for exactly the same set of co-changes.* Therefore, one cannot claim that only one form of coupling out of structural and semantic coupling can better predict co-change as done in [156] where it was stated that conceptual coupling metrics proved to be better predictors for classes impacted by changes compared to structural coupling metrics.

Their findings were not supported in a recent study by Abdeen *et al.* [1]. Our contributions statistically confirm the results in their study. They performed inter-system and intra-system change impact prediction using structural, semantic dependencies and a combination of both and compared results. Their results show that using semantic coupling produces better recall values, in particular, in the intra-system scenario. They stated the addition of semantic coupling data adds extra information that deals with the complexity of structural dependencies in the learning phase. On the other hand, they identified that using structural dependencies or a combination of both types of dependencies out-performs semantic dependencies.

C5 – Computational efficiency. Prior to this study, no study had investigated the efficiency gained when computing semantic coupling between classes using only their identifiers. This Thesis presents an extensive comparison of identifier-based methods to a corpora-based method of semantic coupling measurement. This has been done in order to identify the most efficient ways of computing semantic coupling of OO software classes. It has been identified that identifier-based techniques adopted are more efficient than the corpora-based technique. This will save researchers effort and time in semantic coupling measurement and also provide guidance along the lines of the development of software maintenance tools.

C6 – Tool chain. With regards to semantic coupling measurement, there had been no tool to automate the process of analysing the corpora of Java classes and parsing their corpora to an information retrieval technique like the vector space model.

This Thesis proposed and implemented a tool written in the Java programming language that automates the extraction of semantic coupling metrics using a corpora based approach has been developed. The tool is also publicly available on the GitHub software repository [2].

Based on the core contributions of this Thesis, those who are envisaged to stand to benefit from the aforementioned research contributions as well as the impact of these findings in the software engineering and maintenance domain as defined in Chapter 1 are follows:

1. **Software testers**: Software testers benefit from this Thesis since they will be able to focus their testing efforts with knowledge about the interplay between software dependencies. The relationship between class dependencies and co-evolution would help in software testing. When changes are made to one software class, other classes that have strong semantic or structural coupling to that class also need to be tested. This is to ensure that the changes in one class do not introduce regression faults in other classes.

   However, this means that testing will need to done more than once – based on structural and semantic coupling. But we have proposed refactoring techniques to increase the overlap between semantic and structural coupling as well as reduce unnoticed dependencies during class coupling based change impact analysis. Increasing the overlap between structural and semantic coupling will reduce testing efforts. This is because testing will only be done ONCE based on structural OR semantic coupling information.

[2]The tool can be downloaded at: https://github.com/najienka/SemanticSimilarityJava

2. **Researchers in empirical software engineering and maintenance**:

   In this Thesis we have identified the need for a tool to automate the process of extracting semantic coupling between classes based on their corpora and using IR techniques in the literature. We developed a scalable tool that can automate this process for OSS java projects of different sizes. The tool also makes the semantic coupling measurement process less convoluted. As a result, researchers in the area of semantic coupling in empirical software engineering benefit from this Thesis by using the publicly available tool chain in computing the semantic coupling between software classes of both large and small OSS projects written in Java. The tool eases the process for non-experts in data mining and information retrieval (IR) techniques, provide a standard unified framework for the extraction process as well as promote better comparison of results in this field.

3. **Software comprehension, evolution and maintenance tool chain developers**:

   Software maintenance tools need to consider semantic, structural dependencies because these types of software relationships are essential in minimizing maintenance efforts, improving software understanding and testing.

   Furthermore, relying only on class names or identifiers for semantic coupling measurement yields reasonable results – and it is much cheaper with regards to computational performance compared to analyzing the corpora (the source code) of class pairs. These findings can be used to guide researchers or developers developing future generations of tools supporting program comprehension and software change impact analysis. The findings will also save time in future semantic coupling research and program comprehension for researchers who are non-experts in data analysis.

   We also proposed refactoring techniques to increase the overlap between semantic and structural coupling as well as reduce unnoticed dependencies during class coupling based change impact analysis. These refactoring techniques can be semi-automated *via* the use of developed plugins for software development environments.

4. **Software maintainers**:

   Structural and semantic dependencies are propagators of ripple effects. However, Briand *et al.* showed that if developers are required to handle a large set of dependencies, they would miss a significant number of them [34]. We have proposed novel ways by which the gap between semantic and structural dependencies (software architecture) can be curtailed. This will help to bridge the model-code gap, architecture monitoring, minimize the number of 'hidden' dependencies not captured during change impact analysis when only one dependency type is analyzed. This will in turn make impact analysis process more efficient.

7.4 Personal achievement

Much has been achieved over the course of this research. Firstly, in order to carry out research on a particular topic, the knowledge of the researcher on that topic must be improved. This was achieved over the course of this research by conducting an extensive review of the literature, staying updated with the latest research and attending relevant consortia and conferences.

This helped to identify gaps in the literature, initiate research collaborations with peers, led to the publication of high quality research papers and in turn derive the objective of this research. Notably, the area of semantic coupling is relatively new with the research being done among a closed circle of researchers. The outcome of the knowledge gained during the course of reviewing the literature on semantic coupling is a tool developed with additional features to improve the measurement of semantic coupling. Our contribution to knowledge in the area of semantic coupling has been contributed in a top ranking software engineering conference.

Secondly, it is important for researchers to collaborate and publicise their work. Through workshops and delivering presentations at seminars, collaborations have been built and lessons learned on tailoring presentations to different audiences based on their understanding of the subject area.

Finally, going through the journey of this research has advanced the ability to read and write critically and has improved interpersonal and presentation skills.

7.5 Research limitations

The threats to the validity of this research are summarized by Thesis Chapters 3, 4, 5 and 6 in Subsections 7.5.1, 7.5.2 and 7.5.3. These threats include; internal, external and construct. They are defined [141] as follows:

- **Internal validity** is defined as the accuracy of the conclusion about the study in this research.
- **External validity** is defined as the generalised validity of the conclusions of the research in this thesis.
- **Construct validity** refers to the degree to which a conclusion can be made following the theoretical constructs on which the approach was based.

7.5.1 External validity

EV1 – Generalisability (Ch4, Ch5, Ch6). One of the threats to the external validity of this is the analysis of only seventy nine OSS projects. Notable, this consists of projects of different sizes and domains. However, we acknowledge that this sample might not be representative of all OSS projects in existence or a different sample of projects and this threatens the generalisability of the results.

EV2 – Commercial software (Ch4, Ch5, Ch6). We have analysed only OSS projects. Therefore, it is imperative to identify whether the same results will apply to commercial software. We have considered this as one of the plans for future research.

7.5.2 Internal validity

IV1 – Class identifier-based semantic coupling measurement (Ch5, Ch6). the measurement of the semantic similarity of classes based on their class identifiers. While this is useful in an analysis of coupling, studying the cohesion of classes will require an analysis of their internal structure. We have also compared the metrics derived from analysing class identifiers with those derived from analysing their corpora in Chapter 3.

IV2 – Non-parametric statistical test (Ch4, Ch5, Ch6). We have adopted a non-parametric statistical test (Spearman's rank correlation) when computing the correlation between the studied coupling types. This test was adopted because it cannot be guaranteed that the degree of co-change or structural or semantic coupling will follow a normal distribution (mean = median = mode). We have also not tested for a poisson distribution. Notwithstanding, we have adopted a very low value for α (≤ 0.01). This is to limit the probability or possibility of identifying a significant relationship by chance [207]. The levels of significance are high and the conclusions derived based on the statistical tests are unlikely to differ.

IV3 – Semantic coupling measurement validation (Ch3, Ch5, Ch6). When computing semantic coupling we have manually inspected all class pairs in smaller projects (≤ 20 classes) to validate the similarity metrics derived from the tools. That may have led to a bias while computing semantic coupling. This is because as the number of classes reduces, the dependency and authorship reduces.

IV4 – Class level of granularity (Ch4, Ch5, Ch6). All the empirical studies in this Thesis have been conducted at the class level of granularity. This is because overall,the measurement of semantic coupling is more affected by the difference in granularity than logical or evolutionary coupling. A previous study has shown that for the semantic dependencies, going from the coarse granularity of classes to the finer granularity of methods results in the reduction of the sizes of the documents [102]. The documents are reduced in terms (and frequency). That is, a corpus for a class is typically much "bigger" than a corpus for a method [102]. For logical coupling some commits do not contain changes made to methods while some do not contain changes made to classes, so there is not way to map changes made to classes and methods. This informs the choice of the class level of granularity.

7.5.3 Construct validity

CV1 – Association rule mining for logical coupling measurement (Ch4, Ch5). We acknowledge the fact that support and confidence values of association rules could produce misleading results [148]. For example, if a Java file A joint-changed 7 times with B and afterwards, A changed alone for other 3 times (B did not change anymore). Although the confidence for the logical coupling $A \rightarrow B$ is 0.7, it may be the case that B does not actually depend on A anymore (e.g., after both files changed together for the 7th and last time, B was removed from the system or the structural link from B to A was removed).

7.6 Future work

In this section, we propose and discuss ideas that will drive the future direction of this research.

7.6.1 Empirical studies

Firstly, in this Thesis we have carried out empirical studies on OO software which are all OSS. As such, it is imperative to carry out the same analysis on software which is not OO or OSS. As part of plans for future work, we intend analysing a different sample of OO software which are commercial and not OSS to determine whether our findings are limited to only OSS software.

Secondly, we have studied only software written in the Java programming language. It will be interesting to investigate software written in other languages (e.g., Python, C++) to identify any divergence in findings. In addition, we have studied the interplay between three software coupling types in pairs (e.g., semantic and logical, structural and semantic, and structural and logical). As part of further empirical research, the triangulation of coupling types will be investigated. For example, identifying the portion of classes per project linked by both a logical and semantic relationship. But not a structural relationship or linked by both a logical and structural relationship but not a semantic relationship.

Thirdly, the change impact analysis domain faces a yet to be resolved challenge. As prior research has identified, "it is noteworthy that this actual impact set can vary because changes can be implemented differently" [181]. In this Thesis, it has been demonstrated that change related classes are usually semantically linked. Thus, a feasible experiment will be to investigate the precision of estimated impact sets based on semantic coupling metrics when compared to distinct change implementations by different developers.

Fourthly, this Thesis has not included an investigation of the impact of different design patterns on the studied coupling types and their relationships. For instance, the strategy design pattern will not identify structural couplings but strategies might be semantically linked. It will worth investigating the link between different design patterns and the coupling types studied in this Thesis.

Lastly, Marcus *et al.* [128] advocated the use of latent semantic indexing (LSI) as the supporting information retrieval technique to extract and analyze the semantic information from the source code and associated documentation with the following assumption:

> “our main assumption is that developers use the same natural language (e.g., English, Romanian, etc.) in writing internal documentation and external documentation.”

However, our studied software sample and results have shown that the reverse is the case. This is because developers are likely to name classes and write comments in their native language. Current information retrieval approaches for text mining (e.g., latent semantic analysis) rely on the building of models based on key topics in the documents upon which conceptual similarity is computed. But such models will not scale when used to analyze documents composed of multiple languages as terms will need to be translated before building a unified model for multiple languages. Prior studies have not resolved the problem of measuring semantic coupling in multi-lingual software projects. Therefore, we believe a feasible research topic for future work will be to investigate and develop techniques for the measurement of semantic coupling between multi-lingual OO software classes.

7.6.2 Tool development

In this Chapter 3, we introduced a tool developed to ease the computation of semantic coupling at the class level of granularity in OO software. As plans for future work, we shall extend the tool to capture fine-grained semantic coupling analysis at the method level of granularity. This will also measure the semantic cohesion of classes to support the proposed refactoring approach in Chapter 6. In addition, we plan to build a user interface for the tool to improve its user friendliness.

Finally, we have proposed extract class refactoring approaches in Chapter 6. These approaches are currently carried out manually. Therefore, it is imperative to provide tool support for semi-automation. We plan to in future develop a tool in form of a plug-in for IDEs (e.g., Eclipse [3]) to semi-automate the proposed refactoring approaches.

7.6.3 Studied domains

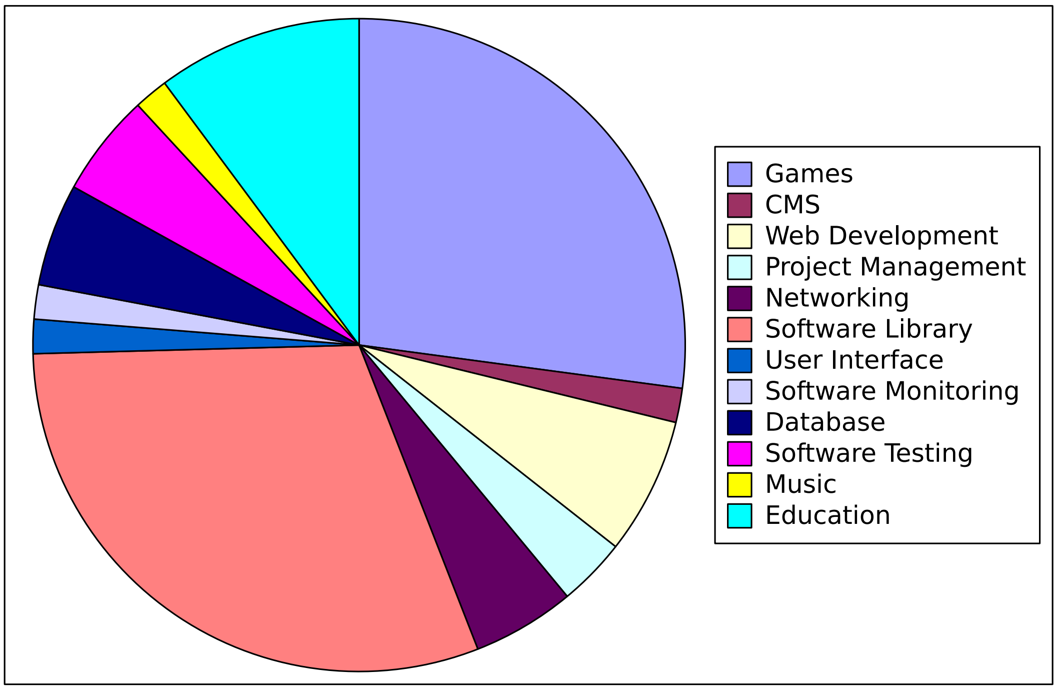


Figure 7.1: Distribution of studied sample of OO software projects by domain

Figure 7.1 depicts a distribution of the studied sample of OO software project by domain. The domains with the highest number of projects are games and software library. As part of the empirical studies for future work, we will investigate the observed results along the various software domains to identify the presence or absence of any interesting patterns or trends.

[3] http://www.eclipse.org/

Bibliography

[1] H. Abdeen, K. Bali, H. Sahraoui, and B. Dufour. Learning Dependency-based Change Impact Predictors Using Independent Change Histories. *Information and Software Technology*, 67:220–235, 2015.

[2] F. B. Abreu and R. Carapuça. Candidate Metrics for Object-Oriented Software Within a Taxonomy Framework. *Journal of Systems and Software*, 26(1):87–96, 1994.

[3] S. N. Ahsan and F. Wotawa. Fault Prediction Capability of Program File's Logical-Coupling Metrics. In *2011 Joint Conference of the* 21st *International Workshop on Software Measurement and* 6th *International Conference on Software Process and Product Measurement (IWSM-MENSURA)*, pages 257–262. IEEE, 2011.

[4] N. Ajienka and A. Capiluppi. Semantic Coupling Between Classes: Corpora or Identifiers? In *Proceedings of the* 10th *ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, page 40. ACM, 2016.

[5] N. Ajienka and A. Capiluppi. Understanding the Interplay between the Logical and Structural Coupling of Software Classes. *Journal of Systems and Software*, 2017.

[6] A. Alali, H. Kagdi, and J. I. Maletic. What's a Typical Commit? A Characterization of Open Source Software Repositories. In *The* 16th *IEEE International*

Conference on Program Comprehension, 2008. ICPC 2008., pages 182–191. IEEE, 2008.

[7] M. Alenezi and K. Magel. Empirical Evaluation of a New Coupling Metric: Combining Structural and Semantic coupling. *International Journal of Computers and Applications*, 36(1), 2014.

[8] E. B. Allen and T. M. Khoshgoftaar. Measuring Coupling and Cohesion: An Information-Theory Approach. In *Sixth International Software Metrics Symposium, 1999. Proceedings.*, pages 119–127. IEEE, 1999.

[9] M. D. Ambros and M. Lanza. Reverse Engineering with Logical Coupling. In 13^{th} *Working Conference on Reverse Engineering, 2006. WCRE'06.*, pages 189–198. IEEE, 2006.

[10] N. Anquetil and T. Lethbridge. Assessing the Relevance of Identifier Names in a Legacy Software System. In *Proceedings of the 1998 Conference of the Centre for Advanced Studies on Collaborative Research*, page 4. IBM Press, 1998.

[11] E. Arisholm, L. C. Briand, and A. Foyen. Dynamic Coupling Measurement for Object-Oriented Software. *IEEE Transactions on Software Engineering.*, 30(8):491–506, 2004.

[12] S. Babu and R. Parvathi. Design Dynamic Coupling Measurement of Distributed Object Oriented Software Using Trace Events. In *Journal of Computer Science*. Citeseer, 2011.

[13] A. Bansal, A. Sachan, and M. Kaur. Analyzing Machine Learning and Statistical Models for Software Change Prediction. *Advances in Computer Science and Information Technology (ACSIT)*, 2015.

[14] M. L. Barnett and R. M. Salomon. Beyond Dichotomy: The Curvilinear Rela-

tionship between Social Responsibility and Financial Performance. *Strategic Management Journal*, 27(11):1101–1122, 2006.

[15] V. R. Basili and D. H. Hutchens. An Empirical Study of a Syntactic Complexity Family. *IEEE Transactions on Software Engineering.*, (6):664–672, 1983.

[16] G. Bavota, A. De Lucia, A. Marcus, and R. Oliveto. A Two-Step Technique for Extract Class Refactoring. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, pages 151–154. ACM, 2010.

[17] G. Bavota, A. De Lucia, A. Marcus, and R. Oliveto. Software Remodularization Based on Structural and Semantic Metrics. In 17^{th} *Working Conference on Reverse Engineering (WCRE), 2010.*, pages 195–204. IEEE, 2010.

[18] G. Bavota, A. De Lucia, A. Marcus, and R. Oliveto. Using Structural and Semantic Measures to Improve Software modularization. *Empirical Software Engineering*, 18(5):901–932, 2013.

[19] G. Bavota, A. De Lucia, A. Marcus, and R. Oliveto. Automating Extract Class Refactoring: an Improved Method and its Evaluation. *Empirical Software Engineering*, 19(6):1617–1664, 2014.

[20] G. Bavota, A. De Lucia, and R. Oliveto. Identifying Extract Class Refactoring Opportunities Using Structural and Semantic cohesion measures. *Journal of Systems and Software*, 84(3):397–414, 2011.

[21] G. Bavota, B. Dit, R. Oliveto, M. Di Penta, D. Poshyvanyk, and A. De Lucia. An Empirical Study on the Developers' Perception of Software Coupling. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 692–701. IEEE Press, 2013.

[22] G. Bavota, M. Gethers, R. Oliveto, D. Poshyvanyk, and A. d. Lucia. Improving Software Modularization via Automated Analysis of Latent Topics and Dependencies. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 23(1):4, 2014.

[23] G. Bavota, R. Oliveto, M. Gethers, D. Poshyvanyk, and A. De Lucia. Methodbook: Recommending Move Method Refactorings via Relational Topic Models. *IEEE Transactions on Software Engineering.*, 40(7):671–694, 2014.

[24] F. Beck and S. Diehl. On the Congruence of Modularity and Code Coupling. In *Proceedings of the* 19th *ACM SIGSOFT Symposium and the* 13th *European Conference on Foundations of Software Engineering*, pages 354–364. ACM, 2011.

[25] F. Beck and S. Diehl. On the Impact of Software Evolution on Software Clustering. *Empirical Software Engineering*, 18(5):970–1004, 2013.

[26] K. Beecher, A. Capiluppi, and C. Boldyreff. Identifying Exogenous Drivers and Evolutionary Stages in FLOSS Projects. *Journal of Systems and Software*, 82(5):739–750, 2009.

[27] K. H. Bennett and V. T. Rajlich. Software Maintenance and Evolution: A Roadmap. In *Proceedings of the Conference on the Future of Software Engineering*, pages 73–87. ACM, 2000.

[28] K. H. Bennett, V. T. Rajlich, and N. Wilde. Software Evolution and the Staged Model of the Software Lifecycle. *Advances in Computers*, 56:1–54, 2002.

[29] P. Bhattacharya, M. Iliofotou, I. Neamtiu, and M. Faloutsos. Graph-based Analysis and prediction for Software Evolution. In 34th *International Conference on Software Engineering (ICSE), 2012.*, pages 419–429. IEEE, 2012.

[30] Bieman, James M and Kang, Byung-Kyoo. Cohesion and Reuse in an Object-Oriented System. In *ACM SIGSOFT Software Engineering Notes*, volume 20, pages 259–262. ACM, 1995.

[31] A. B. Binkley and S. R. Schach. Validation of the Coupling Dependency Metric as a Predictor of Run-Time Failures and Maintenance Measures. In *Proceedings of the 20th International Conference on Software Engineering*, pages 452–455. IEEE Computer Society, 1998.

[32] L. C. Briand, J. W. Daly, and J. K. Wust. A Unified Framework for Coupling Measurement in Object-Oriented Systems. *IEEE Transactions on Software Engineering.*, 25(1):91–121, 1999.

[33] L. C. Briand, S. Morasca, and V. R. Basili. Property-Based Software Engineering measurement. *IEEE Transactions on Software Engineering.*, 22(1):68–86, 1996.

[34] L. C. Briand, J. Wuest, and H. Lounis. Using Coupling Measurement for Impact Analysis in Object-Oriented Systems. In *IEEE International Conference on Software Maintenance, 1999.(ICSM'99) Proceedings.*, pages 475–482. IEEE, 1999.

[35] F. Brito e Abreu, G. Pereira, and P. Sousa. A Coupling-Guided Cluster Analysis Approach to Reengineer the Modularity of Object-Oriented Systems. In *Proceedings of the Fourth European Conference on Software Maintenance and Reengineering, 2000.*, pages 13–22. IEEE, 2000.

[36] F. P. Brooks Jr. *The Mythical Man-Month: Essays on Software Engineering, Anniversary Edition, 2/E.* Pearson Education India, 1995.

[37] Z. Budimlic and K. Kennedy. Optimizing Java: Theory and Practice. *Concurrency: Practice and Experience*, 9(6):445–463, 1997.

[38] E. Burd and M. Munro. An Initial Approach Towards Measuring and Characterising Software Evolution. In 6th *Working Conference on Reverse Engineering, 1999. Proceedings.* , pages 168–174. IEEE, 1999.

[39] G. Canfora, M. Ceccarelli, L. Cerulo, and M. Di Penta. Using Multivariate Time Series and Association Rules to Detect Logical Change coupling: An Empirical Study. In *IEEE International Conference on Software Maintenance (ICSM), 2010.*, pages 1–10. IEEE, 2010.

[40] A. Capiluppi and C. Boldyreff. Identifying and Improving Reusability Based on Coupling Patterns. In *High Confidence Software Reuse in Large Systems*, pages 282–293. Springer, 2008.

[41] Capiluppi, Andrea and Boldyreff, Cornelia and Stol, Klaas-Jan. Successful Reuse of Software Components: A Report From the Open Source Perspective. In *Open Source Systems: Grounding Research*, pages 159–176. Springer, 2011.

[42] B. Caprile and P. Tonella. Restructuring Program Identifier Names. In *icsm*, pages 97–107, 2000.

[43] S. M. Chandrika, E. S. Babu, and N. Srikanth. Conceptual Cohesion of Classes in Object Oriented Systems. *International Journal of Computer Science and Telecommunications*, 2(4):38–44, 2011.

[44] Chapin, Ned and Hale, Joanne E and Khan, Khaled Md and Ramil, Juan F and Tan, Wui-Gee. Types of Software Evolution and Software Maintenance. *Journal of Software Maintenance and evolution: Research and Practice*, 13(1):3–30, 2001.

[45] S. R. Chidamber, D. P. Darcy, and C. F. Kemerer. Managerial Use of Metrics for Object-Oriented Software: An Exploratory Analysis. *IEEE Transactions on Software Engineering.*, 24(8):629–639, 1998.

[46] S. R. Chidamber and C. F. Kemerer. A Metrics Suite for Object Oriented Design. *IEEE Transactions on Software Engineering.*, 20(6):476–493, 1994.

[47] D. Coleman, D. Ash, B. Lowther, and P. Oman. Using Metrics to Evaluate Software System Maintainability. *Computer*, 27(8):44–49, 1994.

[48] K. Collins-Thompson, G. Frishkoff, and S. Crossley. Definition Response Scoring with Probabilistic Ordinal Regression. In *Proceedings of the International Conference on Computers in Education*, volume 6, pages 101–105, 2012.

[49] D. Colquhoun. An Investigation of the False Discovery Rate and the Misinterpretation of p-values. *Royal Society Open Science*, 1(3):140216, 2014.

[50] C. Corley and R. Mihalcea. Measuring the Semantic Similarity of Texts. In *Proceedings of the ACL Workshop on Empirical Modeling of Semantic Equivalence and Entailment*, pages 13–18. Association for Computational Linguistics, 2005.

[51] W. Coster and D. Kauchak. Learning to Simplify Sentences Using Wikipedia. In *Proceedings of the workshop on monolingual text-to-text generation*, pages 1–9. Association for Computational Linguistics, 2011.

[52] D. Cruz, T. Wieland, and A. Ziegler. Evaluation Criteria for Free/Open Source Software Products Based on Project Analysis. *Software Process: Improvement and Practice*, 11(2):107–122, 2006.

[53] M. D'Ambros, M. Lanza, and M. Lungu. The Evolution Radar: Visualizing Integrated Logical Coupling Information. In *Proceedings of the 2006 International Workshop on Mining Software Repositories*, pages 26–32. ACM, 2006.

[54] M. D'Ambros, M. Lanza, and M. Lungu. Visualizing Co-change Information with the Evolution Radar. *IEEE Transactions on Software Engineering.*, 35(5):720–735, 2009.

[55] M. D'Ambros, M. Lanza, and R. Robbes. On the Relationship between Change Coupling and Software Defects. In 16^{th} *Working Conference on Reverse Engineering, 2009. WCRE'09.* , pages 135–144, Lille, France, 2009. IEEE.

[56] A. Dragomir, M. F. Harun, and H. Lichter. On Bridging the Gap between Practice and Vision for Software Architecture reconstruction and evolution: A toolbox perspective. In *Proceedings of the WICSA 2014 Companion Volume*, page 10. ACM, 2014.

[57] R. English and C. M. Schweik. Identifying Success and Tragedy of FLOSS Commons: A Preliminary Classification of Sourceforge. net projects. In *First International Workshop on Emerging Trends in FLOSS Research and Development, 2007. FLOSS'07.* , pages 11–11. IEEE, 2007.

[58] G. Erkan and D. R. Radev. LexRank: Graph-based Lexical Centrality as Salience in Text Summarization. *Journal of Artificial Intelligence Research*, 22:457–479, 2004.

[59] N. Fenton and J. Bieman. *Software Metrics: A Rigorous and Practical Approach.* CRC Press, 2014.

[60] N. E. Fenton and N. Ohlsson. Quantitative Analysis of Faults and Failures in a Complex Software System. *IEEE Transactions on Software Engineering.*, 26(8):797–814, 2000.

[61] J. Fernandez-Ramil, A. Lozano, M. Wermelinger, and A. Capiluppi. Empirical Studies of Open Source Evolution. In *Software Evolution*, pages 263–288. Springer, 2008.

[62] A. Field. *Discovering statistics using SPSS.* Sage publications, 2009.

[63] D. Flanagan. *Java in a Nutshell.* O'Reilly Media, Inc., 2005.

[64] B. Fluri, H. C. Gall, and M. Pinzger. Fine-Grained Analysis of Change Couplings. In *Fifth IEEE International Workshop on Source Code Analysis and Manipulation, 2005.* , pages 66–74. IEEE, 2005.

[65] M. Fowler and K. Beck. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley Professional, 1999.

[66] H. Gall, K. Hajek, and M. Jazayeri. Detection of Logical Coupling Based on Product Release History. In *Proceedings., International Conference on Software Maintenance, 1998.*, pages 190–198. IEEE, 1998.

[67] H. Gall, M. Jazayeri, and J. Krajewski. CVS Release History Data for Detecting Logical Couplings. In *Sixth International Workshop on Principles of Software Evolution, 2003.*, pages 13–23. IEEE, 2003.

[68] E. Gamma. *Design Patterns: Elements of Reusable Object-Oriented Software.* Pearson Education India, 1995.

[69] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Abstraction and Reuse of Object-Oriented Design. In *European Conference on Object-Oriented Programming*, pages 406–431. Springer, 1993.

[70] M. M. Geipel and F. Schweitzer. Software Change Dynamics: Evidence from 35 Java Projects. In *Proceedings of the the* 7^{th} *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, pages 269–272. ACM, 2009.

[71] M. M. Geipel and F. Schweitzer. The Link between Dependency and Cochange: Empirical Evidence. *IEEE Transactions on Software Engineering.*, 38(6):1432–1444, 2012.

[72] D. M. German. An Empirical Study of Fine-Grained Software Modifications. *Empirical Software Engineering*, 11(3):369–393, 2006.

[73] M. Gethers, A. Aryani, and D. Poshyvanyk. Combining Conceptual and Domain-based Couplings to Detect Database and Code Dependencies. In *IEEE* 12th *International Working Conference on Source Code Analysis and Manipulation (SCAM), 2012.*, pages 144–153. IEEE, 2012.

[74] M. Gethers, B. Dit, H. Kagdi, and D. Poshyvanyk. Integrated Impact Analysis for Managing Software Changes. In 34th *International Conference on Software Engineering (ICSE), 2012.*, pages 430–440. IEEE, 2012.

[75] M. Gethers and D. Poshyvanyk. Using Relational Topic Models to Capture Coupling Among Classes in Object-Oriented Software Systems. In *IEEE International Conference on Software Maintenance (ICSM), 2010.*, pages 1–10. IEEE, 2010.

[76] T. Gilb. *Software Metrics*. Winthrop Publishers, 1977.

[77] N. Golafshani. Understanding Reliability and Validity in Qualitative Research. *The Qualitative Report*, 8(4):597–606, 2003.

[78] G. Gousios, M. Pinzger, and A. v. Deursen. An Exploratory Study of the Pull-Based Software Development Model. In *Proceedings of the* 36th *International Conference on Software Engineering*, pages 345–355. ACM, 2014.

[79] S. Haefliger, G. Von Krogh, and S. Spaeth. Code Reuse in Open Source Software. *Management Science*, 54(1):180–193, 2008.

[80] M. Hahsler, B. Gruen, K. Hornik, and M. M. Hahsler. The `arules` Package. *methods*, 15:1, 2006.

[81] M. Hahsler, B. Grün, and K. Hornik. A Computational Environment for Mining Association Rules and Frequent Item Sets. 2005.

[82] M. Hahsler, B. Grün, and K. Hornik. Introduction to Arules–Mining Association Rules and Frequent Item Sets. *SIGKDD Explor*, 2(4), 2007.

[83] M. Hahsler and K. Hornik. Building on the `arules` Infrastructure for Analyzing Transaction Data with R. In *Advances in Data Analysis*, pages 449–456. Springer, 2007.

[84] G. A. Hall, W. Tao, and J. C. Munson. Measurement and Validation of Module Coupling Attributes. *Software Quality Journal*, 13(3):281–296, 2005.

[85] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The WEKA Data Mining Software: An Update. *ACM SIGKDD Explorations Newsletter*, 11(1):10–18, 2009.

[86] T. Hall and P. Wernick. Program Slicing Metrics and Evolvability: An Initial Study. In *IEEE International Workshop on Software Evolvability, 2005.*

[87] N. Hanakawa. Visualization for Software Evolution based on Logical Coupling and Module Coupling. In 14th *Asia-Pacific Software Engineering Conference, 2007. APSEC 2007.* , pages 214–221. IEEE, 2007.

[88] A. E. Hassan. The Road Ahead for Mining Software Repositories. In *Frontiers of Software Maintenance, 2008. FoSM 2008.*, pages 48–57. IEEE, 2008.

[89] A. E. Hassan and R. C. Holt. Predicting Change Propagation in Software Systems. In *20th IEEE International Conference on Software Maintenance, 2004. Proceedings.* , pages 284–293. IEEE, 2004.

[90] B. Henderson-Sellers, L. L. Constantine, and I. M. Graham. Coupling and Cohesion (Towards a Valid Metrics Suite for Object-Oriented Analysis and Design). *Object Oriented Systems*, 3(3):143–158, 1996.

[91] S. Henry and D. Kafura. Software Structure Metrics based on Information Flow. *IEEE Transactions on Software Engineering.*, (5):510–518, 1981.

[92] E. Hill, L. Pollock, and K. Vijay-Shanker. Exploring the Neighborhood with Dora to Expedite Software Maintenance. In *Proceedings of the* 22nd

IEEE/ACM International Conference on Automated Software Engineering, pages 14–23. ACM, 2007.

[93] A. Hindle, D. M. German, and R. Holt. What Do Large Commits Tell Us?: A Taxonomical Study of Large Commits. In *Proceedings of the 2008 International Working Conference on Mining Software Repositories*, pages 99–108. ACM, 2008.

[94] G. S. Howard and S. E. Maxwell. Correlation between Student Satisfaction and Grades: A Case of Mistaken Causation? *Journal of Educational Psychology*, 72(6):810, 1980.

[95] J. Howison, M. Conklin, and K. Crowston. FLOSSmole: A Collaborative Repository for FLOSS Research Data and Analyses. *International Journal of Information Technology and Web Engineering (IJITWE)*, 1(3):17–26, 2006.

[96] A. Huang. Similarity Measures for Text Document Clustering. In *Proceedings of the Sixth New Zealand Computer Science Research Student Conference (NZCSRSC2008), Christchurch, New Zealand*, pages 49–56, 2008.

[97] H. Kabaili, R. K. Keller, and F. Lustman. Cohesion as Changeability Indicator in Object-Oriented Systems. In *Fifth European Conference on Software Maintenance and Reengineering, 2001.* , pages 39–46. IEEE, 2001.

[98] D. Kafura and S. Henry. Software Quality Metrics based on Interconnectivity. *Journal of Systems and Software*, 2(2):121–131, 1981.

[99] H. Kagdi. Improving Change Prediction With Fine-Grained Source Code Mining. In *Proceedings of the Twenty-Second IEEE/ACM International Conference on Automated Software Engineering*, pages 559–562. ACM, 2007.

[100] H. Kagdi, M. L. Collard, and J. I. Maletic. A Survey and Taxonomy of Approaches for Mining Software Repositories in the Context of Software Evolu-

tion. *Journal of Software Maintenance and Evolution: Research and Practice*, 19(2):77–131, 2007.

[101] H. Kagdi, M. L. Collard, and J. I. Maletic. A Survey and Taxonomy of Approaches for Mining Software Repositories in the Context of Software Evolution. *J. Softw. Maint. Evol.*, 19(2):77–131, Mar. 2007.

[102] H. Kagdi, M. Gethers, and D. Poshyvanyk. Integrating Conceptual and Logical Couplings for Change Impact Analysis in Software. *Empirical Software Engineering*, 18(5):933–969, 2013.

[103] H. Kagdi, M. Gethers, D. Poshyvanyk, and M. L. Collard. Blending Conceptual and Evolutionary Couplings to Support Change Impact Analysis in Source Code. In 17*th Working Conference on Reverse Engineering (WCRE), 2010.*, pages 119–128. IEEE, 2010.

[104] H. Kagdi and J. I. Maletic. Combining Single-Version and Evolutionary Dependencies for Software-Change Prediction. In *Fourth International Workshop on Mining Software Repositories, 2007. ICSE Workshops MSR'07.* , pages 17–17. IEEE, 2007.

[105] M. Kajko-Mattsson. *Corrective Maintenance Maturity Model: Problem Management.* Stockholm University, Department of Computer and Systems Sciences, 2001.

[106] M. Kajko-Mattsson, S. Forssander, and U. Olsson. Corrective Maintenance Maturity Model (CM 3): Maintainer's Education and Training. In *Proceedings of the* 23rd *International Conference on Software Engineering, 2001. ICSE 2001.*, pages 610–619. IEEE, 2001.

[107] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, and D. Damian. An In-Depth Study of the Promises and Perils of Mining GitHub. *Empirical Software Engineering*, 21(5):2035–2071, 2016.

[108] R. Kenett and S. Salini. Relative Linkage Disequilibrium: A New Measure for Association Rules. In *Industrial Conference on Data Mining*, pages 189–199. Springer, 2008.

[109] V. Kešelj, F. Peng, N. Cercone, and C. Thomas. N-Gram-based Author Profiles for Authorship Attribution. In *Proceedings of the Conference Pacific Association for Computational Linguistics, PACLING*, volume 3, pages 255–264, 2003.

[110] B. A. Kitchenham, L. M. Pickard, and S. J. Linkman. An Evaluation of Some Design Metrics. *Software Engineering Journal*, 5(1):50–58, 1990.

[111] J. Knudsen and J. Director-Zukowski. *Wireless Java: Developing with J2ME.* APress LP, 2003.

[112] A. G. Koru and H. Liu. Identifying and Characterizing Change-Prone Classes in two Large-Scale Open-Source Products. *Journal of Systems and Software*, 80(1):63–73, 2007.

[113] A. Kuhn, S. Ducasse, and T. Gírba. Semantic Clustering: Identifying Topics in Source Code. *Information and Software Technology*, 49(3):230–243, 2007.

[114] D. Kung, J. Gao, P. Hsia, F. Wen, Y. Toyoshima, and C. Chen. Change Impact Identification in Object Oriented Software Maintenance. In *International Conference on Software Maintenance, 1994. Proceedings.,* , pages 202–211. IEEE, 1994.

[115] M. M. Lehman. Programs, Life Cycles, and Laws of Software Evolution. *Proceedings of the IEEE*, 68(9):1060–1076, 1980.

[116] J. Lewis and J. Ritchie. Generalising from Qualitative Research. *Qualitative Research Practice: A Guide for Social Science Students and Researchers*, 2:347–362, 2003.

[117] H. Li. A Novel Coupling Metric for Object-Oriented Software Systems. In *IEEE International Symposium on Knowledge Acquisition and Modeling Workshop, 2008. KAM Workshop 2008.*, pages 609–612. IEEE, 2008.

[118] B. P. Lientz and E. B. Swanson. Software Maintenance Management: A Study of the Maintenance of Computer Application Software in 487 Data Processing Organizations. 1980.

[119] B. P. Lientz, E. B. Swanson, and G. E. Tompkins. Characteristics of Application Software Maintenance. *Communications of the ACM*, 21(6):466–471, 1978.

[120] Y. Liu and A. Milanova. Static Analysis for Dynamic Coupling Measures. In *Proceedings of the 2006 Conference of the Center for Advanced Studies on Collaborative Research*, page 10. IBM Corp., 2006.

[121] M. Lorenz and J. Kidd. *Object-Oriented Software Metrics: A Practical Guide.* Prentice-Hall, Inc., 1994.

[122] H. Lu, Y. Zhou, B. Xu, H. Leung, and L. Chen. The Ability of Object-Oriented Metrics to Predict Change-Proneness: A Meta-Analysis. *Empirical Software Engineering*, 17(3):200–242, 2012.

[123] S. Maheshwari, J. Agrawal, and S. Sharma. A New Approach for Classification of Highly Imbalanced Datasets using Evolutionary Algorithms. *International Journal of Scientific & Engineering Research*, 2(7):1–5, 2011.

[124] J. I. Maletic and A. Marcus. Supporting Program Comprehension Using Semantic and Structural Information. In *Proceedings of the 23rd International Conference on Software Engineering*, pages 103–112. IEEE Computer Society, 2001.

[125] R. Malhotra, A. Bansal, and S. Jajoria. An Automated Tool for Generating Change Report from Open-source Software. In *International Conference on*

Advances in Computing, Communications and Informatics (ICACCI), 2016., pages 1576–1582. IEEE, 2016.

[126] R. Malhotra and A. J. Bansal. Cross Project Change Prediction Using Open Source Projects. In *International Conference on Advances in Computing, Communications and Informatics (ICACCI, 2014.*, pages 201–207. IEEE, 2014.

[127] H. Malik and A. E. Hassan. Supporting Software Evolution Using Adaptive Change Propagation Heuristics. In *IEEE International Conference on Software Maintenance, 2008. ICSM 2008.* , pages 177–186. IEEE, 2008.

[128] A. Marcus, J. I. Maletic, and A. Sergeyev. Recovery of Traceability Links between Software Documentation and Source Code. *International Journal of Software Engineering and Knowledge Engineering*, 15(05):811–836, 2005.

[129] A. Marcus and D. Poshyvanyk. The Conceptual Cohesion of Classes. In 21^{st} *IEEE International Conference on Software Maintenance (ICSM'05)*, pages 133–142. IEEE, 2005.

[130] A. Marcus, A. Sergeyev, V. Rajlich, J. Maletic, et al. An Information Retrieval Approach to Concept Location in Source Code. In 11^{th} *Working Conference on Reverse Engineering, 2004. Proceedings.*, pages 214–223. IEEE, 2004.

[131] M. Martin and R. C. Martin. *Agile Principles, Patterns, and Practices in C#.* Pearson Education, 2006.

[132] J. A. McDermid. *Software Engineer's Reference Book.* Elsevier, 1991.

[133] P. McMinn, C. Wright, C. Kinneer, C. McCurdy, M. Camara, and G. Kapfhammer. SchemaAnalyst: Search-Based Test Data Generation for Relational Database Schemas. In *Proceedings of the* 32^{nd} *International Conference on Software Maintenance and Evolution*, 2016.

[134] P. Mcnamee and J. Mayfield. Character N-Gram Tokenization for European Language Text Retrieval. *Information retrieval*, 7(1-2):73–97, 2004.

[135] P. D. McNicholas, T. B. Murphy, and M. O'Regan. Standardising the Lift of an Association Rule. *Computational Statistics & Data Analysis*, 52(10):4712–4721, 2008.

[136] T. Mens. *Introduction and Roadmap: History and Challenges of Software Evolution*. Springer, 2008.

[137] T. Mens, M. Wermelinger, S. Ducasse, S. Demeyer, R. Hirschfeld, and M. Jazayeri. Challenges in Software Evolution. In *Principles of Software Evolution, Eighth International Workshop on*, pages 13–22. IEEE, 2005.

[138] S. J. Metsker. *The Design Patterns Java Workbook*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[139] V. Midha and P. Palvia. Factors Affecting the Success of Open Source Software. *Journal of Systems and Software*, 85(4):895–905, 2012.

[140] A. Mitchell and J. F. Power. Using Object-Level Run-Time Metrics to Study Coupling between Objects. In *Proceedings of the 2005 ACM symposium on Applied computing*, pages 1456–1462. ACM, 2005.

[141] M. Mitchell and J. Jolley. *Research Design Explained*. Cengage Learning, 2012.

[142] M. Mondal, C. K. Roy, and K. A. Schneider. Connectivity of Co-Changed Method Groups: A Case Study on Open Source Systems. In *Proceedings of the 2012 Conference of the Center for Advanced Studies on Collaborative Research*, pages 205–219. IBM Corp., 2012.

[143] S. Morasca and L. C. Briand. Towards a Theoretical Framework for Measuring Software Attributes. In *Fourth International Software Metrics Symposium, 1997. Proceedings.,*, pages 119–126. IEEE, 1997.

[144] A. Mubarak. *An Empirical Study of Package Coupling in Java Open-Source.* PhD thesis, Brunel University, School of Information Systems, Computing and Mathematics, 2010.

[145] I. Navarro, N. Leveson, and K. Lunqvist. Semantic Decoupling: Reducing the Impact of Requirement Changes. *Requirements Engineering*, 15(4):419–437, 2010.

[146] A. J. Offutt, M. J. Harrold, and P. Kolte. A Software Metric System for Module Coupling. *Journal of Systems and Software*, 20(3):295–308, 1993.

[147] G. A. Oliva and M. Gerosa. Experience Report: How Do Structural Dependencies Influence Change Propagation? An Empirical Study. In *Proceedings of the* 26th *IEEE International Symposium on Software Reliability Engineering*, 2015.

[148] G. A. Oliva and M. A. Gerosa. On the Interplay Between Structural and Logical Dependencies in Open-Source Software. In 25th *Brazilian Symposium on Software Engineering (SBES), 2011.*, pages 144–153. IEEE, 2011.

[149] G. A. Oliva and M. A. Gerosa. A Method for the Identification of Logical Dependencies. In *IEEE Seventh International Conference on Global Software Engineering Workshops (ICGSEW), 2012.*, pages 70–72. IEEE, 2012.

[150] G. A. Oliva, F. W. Santana, M. A. Gerosa, and C. R. De Souza. Towards a Classification of Logical Dependencies Origins: A Case Study. In *Proceedings of the* 12th *International Workshop on Principles of Software Evolution and the* 7th *annual ERCIM Workshop on Software Evolution*, pages 31–40. ACM, 2011.

[151] R. R. Pagano. *Understanding Statistics in the Behavioral Sciences.* Wadsworth-Thomson Learning, Australia;United Kingdom;, 6th edition, 2001.

[152] D. L. Parnas. Software Aging. In *Proceedings of the* 16th *International Conference on Software Engineering*, pages 279–287. IEEE Computer Society Press, 1994.

[153] A. Perdicoúlis. Correlation and Causality. *Oestros*, pages 2–5, 2013.

[154] M. Petrenko and V. Rajlich. Variable Granularity for Improving Precision of Impact Analysis. In *IEEE* 17th *International Conference on Program Comprehension, 2009. ICPC'09.* , pages 10–19. IEEE, 2009.

[155] D. Poshyvanyk and A. Marcus. The Conceptual Coupling Metrics for Object-Oriented Systems. In *22nd IEEE International Conference on Software Maintenance, 2006. ICSM'06.*, pages 469–478. IEEE, 2006.

[156] D. Poshyvanyk, A. Marcus, R. Ferenc, and T. Gyimóthy. Using Information Retrieval Based Coupling Measures for Impact Analysis. *Empirical Software Engineering*, 14(1):5–32, 2009.

[157] L. Prechelt. An Empirical Comparison of C, C++, Java, Perl, Python, Rexx and Tcl. *IEEE Computer*, 33(10):23–29, 2000.

[158] R. S. Pressman. *Software Engineering: A Practitioner's Approach.* Palgrave Macmillan, 2005.

[159] Queille, Jean-Plene and Voidrot, Jean-Franmis and Wilde, Norman and Munro, Malcolm. The Impact Analysis Task in Software Maintenance: A Model and A Case Study. In *Proceedings., International Conference on Software Maintenance, 1994.*, pages 234–242. IEEE, 1994.

[160] A. Qusef, G. Bavota, R. Oliveto, A. De Lucia, and D. Binkley. Scotch: Test-to-code Traceability Using Slicing and Conceptual Coupling. In 27th *IEEE International Conference on Software Maintenance (ICSM), 2011.*, pages 63–72. IEEE, 2011.

[161] A. Rainer and S. Gale. Evaluating the Quality and Quantity of Data on Open Source Software Projects. In *Proceedings of the* 1st *International Conference on Open Source Software*, 2005.

[162] X. Ren, B. G. Ryder, M. Stoerzer, and F. Tip. Chianti: A Change Impact Analysis Tool for Java Programs. In 27th *International Conference on Software Engineering, 2005. ICSE 2005. Proceedings.* , pages 664–665. IEEE, 2005.

[163] M. Revelle, M. Gethers, and D. Poshyvanyk. Using Structural and Textual Information to Capture Feature Coupling in Object-Oriented Software. *Empirical Software Engineering*, 16(6):773–811, 2011.

[164] M. Riaz, T. Mahmood, and M. Arslan. Non-Parametric Versus Parametric Methods in Environmental Sciences. *Bulletin of Envi*, 2016.

[165] J. Rilling, R. Witte, D. Gaševic, and J. Z. Pan. Semantic Technologies in System Maintenance (STSM 2008). In *The 16th IEEE International Conference on Program Comprehension, 2008. ICPC 2008.* , pages 279–282. IEEE, 2008.

[166] M. P. Robillard. Automatic Generation of Suggestions for Program Investigation. In *ACM SIGSOFT Software Engineering Notes*, volume 30, pages 11–20. ACM, 2005.

[167] G. Robles, J. M. González-Barahona, D. Izquierdo-Cortazar, and I. Herraiz. Tools and Datasets for Mining Libre Software Repositories. *Multi-Disciplinary Advancement in Open Source Software and Processes, IGI Global, Hershey*, pages 24–42, 2011.

[168] G. Robles, S. Koch, J. M. GonZÁlEZ-BARAHonA, and J. Carlos. Remote Analysis and Measurement of Libre Software Systems by Means of the CVS-AnalY Tool. In *Proceedings of the* 2nd *ICSE Workshop on Remote Analysis and Measurement of Software Systems (RAMSS)*, pages 51–56, 2004.

[169] B. A. Romo, A. Capiluppi, and T. Hall. Filling the Gaps of Development Logs and Bug Issue Data. In *Proceedings of The International Symposium on Open Collaboration*, page 8. ACM, 2014.

[170] B. G. Ryder and F. Tip. Change Impact Analysis for Object-Oriented Programs. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 46–53. ACM, 2001.

[171] I. Samoladas, L. Angelis, and I. Stamelos. Survival Analysis on the Duration of Open Source Projects. *Information and Software Technology*, 52(9):902–922, 2010.

[172] R. Sarikaya, A. Gravano, and Y. Gao. Rapid Language Model Development Using External Resources for New Spoken Dialog Domains. In *ICASSP (1)*, pages 573–576, 2005.

[173] C. M. Schweik, R. English, Q. Paienjton, and S. Haire. Success and Abandonment in Open Source Commons: Selected Findings from an Empirical Study of Sourceforge. net projects. In *Proceedings of the Sixth International Conference on Open Source Systems (OSS 2010) Workshops*, 2010.

[174] A. R. Sharafat and L. Tahvildari. Change Prediction in Object-Oriented Software Systems: A Probabilistic Approach. *Journal of Software*, 3(5):26–39, 2008.

[175] T. Sharma and G. Suryanarayana. Augur: Incorporating Hidden Dependencies and Variable Granularity in Change Impact Analysis. In *IEEE* 16^{th} *International Working Conference on Source Code Analysis and Manipulation (SCAM), 2016* , pages 73–78. IEEE, 2016.

[176] M. Shepperd and D. Ince. *Derivation and Validation of Software Metrics.* Oxford University Press, Inc., 1993.

[177] G. Singh and M. D. Ahmed. Effect of Coupling on Change in Open Source Java Systems. In *Proceedings of the Australasian Computer Science Week Multiconference*, page 22. ACM, 2017.

[178] I. Sommerville. Software Engineering. International Computer Science Series. *ed: Addison Wesley*, 2004.

[179] G. Sridhara, E. Hill, L. Pollock, and K. Vijay-Shanker. Identifying Word Relations in Software: A Comparative Study of Semantic Similarity Tools. In *The* 16^{th} *IEEE International Conference on Program Comprehension, 2008. ICPC 2008.*, pages 123–132. IEEE, 2008.

[180] R. E. Stake. *The Art of Case Study Research.* Sage, 1995.

[181] X. Sun, B. Li, H. Leung, B. Li, and J. Zhu. Static Change Impact Analysis Techniques: A Comparative Study. *Journal of Systems and Software*, 109:137–149, 2015.

[182] X. Sun, B. Li, C. Tao, W. Wen, and S. Zhang. Change Impact Analysis Based on a Taxonomy of Change Types. In *Computer Software and Applications Conference (COMPSAC), 2010 IEEE* 34^{th} *Annual*, pages 373–382. IEEE, 2010.

[183] C. L. Tan, S. Y. Sung, Z. Yu, and Y. Xu. Text Retrieval from Document Images based on N-Gram Algorithm. In *PRICAI Workshop on Text and Web Mining*, pages 1–12. Citeseer, 2000.

[184] L. Tokuda and D. Batory. Evolving Object-Oriented Designs with Refactorings. *Automated Software Engineering*, 8(1):89–120, 2001.

[185] N. Tsantalis, A. Chatzigeorgiou, and G. Stephanides. Predicting the Probability of Change in Object-Oriented Systems. *IEEE Transactions on Software Engineering.*, 31(7):601–614, 2005.

[186] P. D. Turney. A Uniform Approach to Analogies, Synonyms, Antonyms, and Associations. In *Proceedings of the* 22nd *International Conference on Computational Linguistics-Volume 1*, pages 905–912. Association for Computational Linguistics, 2008.

[187] B. Újházi, R. Ferenc, D. Poshyvanyk, and T. Gyimóthy. New Conceptual Coupling and Cohesion Metrics for Object-Oriented Systems. In *10th IEEE Working Conference on Source Code Analysis and Manipulation (SCAM), 2010.*, pages 33–42. IEEE, 2010.

[188] M. Van Antwerp and G. Madey. Advances in the Sourceforge Research Data Archive. In *Workshop on Public Data about Software Development (WoPDaSD) at The* 4th *International Conference on Open Source Systems, Milan, Italy*, 2008.

[189] H. Van Vliet, H. Van Vliet, and J. Van Vliet. *Software Engineering: Principles and Practice*, volume 3. Wiley, 1993.

[190] R. Vanciu and V. Rajlich. Hidden Dependencies in Software Systems. In *IEEE International Conference on Software Maintenance (ICSM), 2010.*, pages 1–10. IEEE, 2010.

[191] B. Verhulst, L. J. Eaves, and P. K. Hatemi. Correlation not Causation: The Relationship between Personality Traits and Political Ideologies. *American journal of political science*, 56(1):34–51, 2012.

[192] B. Vidgen and T. Yasseri. P-values: Misunderstood and Misused. *arXiv preprint arXiv:1601.06805*, 2016.

[193] A. Von Mayrhauser and A. M. Vans. Program Comprehension During Software Maintenance and Evolution. *Computer*, 28(8):44–55, 1995.

[194] R. Wang, R. Huang, and B. Qu. Network-Based Analysis of Software Change Propagation. *The Scientific World Journal*, 2014, 2014.

[195] I. Wiese, R. Kuroda, R. Ré, R. Bulhões, G. Oliva, and M. Gerosa. Do Historical Metrics and Developers Communication Aid to Predict Change Couplings? *Latin America Transactions, IEEE (Revista IEEE America Latina)*, 13(6):1979–1988, 2015.

[196] I. S. Wiese, R. T. Kuroda, R. Re, G. A. Oliva, and M. A. Gerosa. An Empirical Study of the Relation Between Strong Change Coupling and Defects Using History and Social Metrics in the Apache Aries Project. In *IFIP International Conference on Open Source Systems*, pages 3–12. Springer, 2015.

[197] N. Wilde and R. Huitt. Maintenance Support for Object Oriented Programs. In *Conference on Software Maintenance, 1991., Proceedings.*, pages 162–170. IEEE, 1991.

[198] F. G. Wilkie and B. A. Kitchenham. Coupling Measures and Change Ripples in C++ Application Software. *Journal of Systems and Software*, 52(2):157–164, 2000.

[199] R. Witte, Q. Li, Y. Zhang, and J. Rilling. Text Mining and Software Engineering: an Integrated Source Code and Document Analysis Approach. *Iet Software*, 2(1):3–16, 2008.

[200] B. R. E. Wright, A. Caspi, T. E. Moffitt, R. A. Miech, and P. A. Silva. Reconsidering the Relationship between SES and Delinquency: Causation but not Correlation. *Criminology*, 37(1):175–194, 1999.

[201] K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella, et al. Titanium: A High-Performance Java Dialect. *Concurrency Practice and Experience*, 10(11-13):825–836, 1998.

[202] A. T. Ying, G. C. Murphy, R. Ng, and M. C. Chu-Carroll. Predicting Source Code Changes by Mining Change History. *IEEE Transactions on Software Engineering.*, 30(9):574–586, 2004.

[203] L. Yu. Understanding Component Co-Evolution with a Study on Linux. *Empirical Software Engineering*, 12(2):123–141, 2007.

[204] L. Yu, A. Mishra, and S. Ramaswamy. Component Co-evolution and Component Dependency: Speculations and Verifications. *Software, IET*, 4(4):252–267, 2010.

[205] Z. Yu and V. Rajlich. Hidden Dependencies in Program Comprehension and Change Propagation. In 9th *International Workshop on Program Comprehension, 2001. IWPC 2001. Proceedings.*, pages 293–299. IEEE, 2001.

[206] Y. Zhao and S. S. Bhowmick. Association rule mining with r. *A Survey Nanyang Technological University, Singapore*, 2015.

[207] Y. Zhao, F. Zhang, E. Shihab, Y. Zou, and A. E. Hassan. How Are Discussions Associated with Bug Reworking?: An Empirical Study on Open Source Projects. In *Proceedings of the* 10th *ACM/IEEE International Symposium on Empirical Software Engineering and Measurement.*

[208] T. Zimmermann, S. Diehl, and A. Zeller. How History Justifies System Architecture (or Not). In *Sixth International Workshop on Principles of Software Evolution, 2003. Proceedings.*, pages 73–83. IEEE, 2003.

[209] T. Zimmermann, R. Premraj, and A. Zeller. Predicting Defects for Eclipse. In *International Workshop on Predictor Models in Software Engineering, 2007. PROMISE'07: ICSE Workshops 2007.*, pages 9–9. IEEE, 2007.

[210] T. Zimmermann, A. Zeller, P. Weissgerber, and S. Diehl. Mining Version Histories to Guide Software Changes. *IEEE Transactions on Software Engineering.*, 31(6):429–445, 2005.

Glossary of software engineering terms

Class A class is a template that defines the attributes and functions of an object.

Coupling Coupling is the intensity of the interconnection between artifacts, such as classes.

Cohesion Cohesion is the degree of how linked or focused the responsibilities of an artifact are, e.g., the functions of a class.

Co-evolution Co-evolution is the degree to which two artifacts are co-changed.

Commit A commit is a change or set of changes to a file or set of files.

Interplay The manner by which two or more things directly influence each other.

Method A method is a procedure or functionality defined and implemented in a class. It defines the role of instances of that class, an object.

Object An object in object-oriented software is an instance of a class. It can contain variables, functions or actions and data structures such as lists, arrays, etc.

Object-oriented software Also known as OO software is software composed of inter-dependent objects.

Open-source software Open-source software (OSS) is software that is publicly available and open to modification, and distribution by anyone.

Outlier A person or thing that is very different from the rest of a group or set.

Software License A software licence is an instrument that dictates the rights associated with the modification and redistribution of software.

Refactoring Refactoring is changing the internal structure of software without altering its outward behaviour.

Repository A software repository is a storage location from which software may be recovered.

Reuse Reuse is the use of already built software to build or advance existing software functionality without rewriting any functionality from scratch.

Revision Another name for "commit", a revision is a change or set of changes to a file or set of files.

Software architecture Software architecture is the definition of the internal structure of a software including its components, their relationships and external parts.

Software metrics Software metrics are measurements used to assess the quality of software projects

Version control system A version control system is one which keeps track of changes or sets of changes or commits of a software project.

Appendix A

Appendix A consists of a summary of the studied sample of OO OSS projects in this Thesis.

A.1 Software dependency metrics

Table 1 summarizes proposed software dependency metrics. The first column states the name of each metric, followed by the study in which they are proposed in the second column, the dependency type[4] concerned in the third column and the year in which the study was published presented in the fourth and last column.

A.2 OSS project sample description

In this Thesis, we have studied 79 distinct OSS projects developed in the `JAVA` programming language and summarized in Table 2. Each project from the studied sample is described briefly below:

1. 2dtetris: java Tetris developed using SCRUM.
2. 4-connect:
3. ahs-scheduling: a java scheduling application built to schedule high school courses.

[4]picked from the three studied dependency types in the Thesis.

Table 1: Summary of software dependency metrics

Metric	Study	Dependency Type	Published
Lines of code (LOC)	Gilb [76]	Structural	1977
Methods per class (NM)	Lorenz and Kidd [121]	Structural	1994
Number of Public Methods per class (NPM)	Lorenz and Kidd [121]	Structural	1994
Number of Variables per class (NV)	Lorenz and Kidd [121]	Structural	1994
Number of Public Variables per class (NPV)	Lorenz and Kidd [121]	Structural	1994
Number of Methods Inherited (NMI)	Lorenz and Kidd [121]	Structural	1994
Number of Methods Overridden (NMO)	Lorenz and Kidd [121]	Structural	1994
Number of Methods Added(NMA)	Lorenz and Kidd [121]	Structural	1994
Average Methods Size (AMS)	Lorenz and Kidd [121]	Structural	1994
Number of times a Class is Reused (NCR)	Lorenz and Kidd [121]	Structural	1994
Number of Friends of class (NF)	Lorenz and Kidd [121]	Structural	1994
Coupling between objects (CBO)	Chidamber *et al.* [46]	Structural	1994
Response for a class (RFC)	Chidamber *et al.* [46]	Structural	1994
Message passing coupling (MPC)	Chidamber *et al.* [46]	Structural	1994
Lack of cohesion (LCOM)	Chidamber *et al.* [46]	Structural	1994
Weighted methods per class (WMC)	Chidamber *et al.* [46]	Structural	1994
Method Inheritance Factor (MIF)	Abreu and Carapuca [2]	Structural	1994
Attribute Inheritance Factor (AIF)	Abreu and Carapuca [2]	Structural	1994
Coupling Factor (CF)	Abreu and Carapuca [2]	Structural	1994
Polymorphism Factor (PF)	Abreu and Carapuca [2]	Structural	1994

Metric	Study	Dependency Type	Published
Method Hiding Factor (MHF)	Abreu and Carapuca [2]	Structural	1994
Attribute Hiding Factor (AHF)	Abreu and Carapuca [2]	Structural	1994
Support and Confidence	Zimmermann *et al.* [208]	Logical	2003
Conceptual coupling of classes (CoCC)	Poshyvanyk and Marcus [155]	Semantic	2006
Conceptual Similarity between Methods (CSM)	Poshyvanyk and Marcus [155]	Semantic	2006
Conceptual similarity between two classes (CSBC)	Poshyvanyk and Marcus [155]	Semantic	2006
Relational Topic based Coupling (RTC)	Gethers and Poshyvanyk [75]	Semantic	2010
Coupling between Object Classes (CCBO)	Ujhazi *et al.* [187]	Semantic	2010

4. aima-java: a java implementation of algorithms from Russell And Norvig's "Artificial Intelligence - A Modern Approach".

5. alexo-chess: java chess engine. It plugs into WinChess, based on UBT.

6. algmusic: algmusic is a java web application that can be used to organize mp3 files.

7. alleywayreinvented: a small java game similar to the old Game Boy game. The core game logic is the same but we are adding lost of other features.

8. alto: alto is the collection of interfaces implemented in the syntelos and syntelos-sx packages [5].

9. amock: amock automatically generates java interaction-based unit tests from system tests.

10. apjava:

11. appletbomberman: bomberman game in a java applet.

[5]The project owners mention that "separation of system interfaces into an independent project aids development"

Table 2: Summary of Sample of OO OSS projects studied

Project ID	Name	# Classes	# Revisions
1	2dtetris	26	203
2	4-connect	12	29
7	ahs-scheduling	20	141
8	aima-java	870	258
10	alexo-chess	142	105
11	algmusic	110	56
12	alleywayreinvented	45	61
13	alto	297	75
14	amock	207	456
18	apjava	23	35
20	appletbomberman	66	64
22	ascrblr	53	53
24	audao	154	21
26	bitlyj	60	46
28	bluecove	389	258
30	castanea	48	33
31	catchnthrow	22	72
41	daedalum	72	26
45	dbmigrate	8	25
51	echo-nest-java-api	45	40
56	fdelimitedtextutilities	14	25
60	fyllgen	123	51
64	geocoder-java	36	61
65	google-voice-java	61	172
66	gorobot	450	32
67	gp-net-radius	31	22
68	guavatools	270	56
69	guitarjava	102	94
71	hobbylinkchecker	193	29
79	jangod	153	36
81	jaque	70	123
84	java-chess-web	143	96
86	java-weather-api	36	34
88	javacoder	22	23
92	javastepbystep	137	81
96	jbal	121	117
97	jbandwidthlog	28	53
99	jease	296	350
103	jeudi-tech-spring	35	41

Project ID	Name	# Classes	# Revisions
107	jiopi	36	47
109	jmemcache	19	24
112	jnoob	65	29
113	jothelo	20	33
115	jprg2-assg	23	71
118	jroguedps	124	72
119	jsbe	11	54
122	jtowerdefense	84	87
123	jugile-util	88	61
124	jutf8search	15	42
127	kryo	148	289
130	lemyriapode	447	292
136	migrator-postgresql	31	35
140	mobs	45	43
141	mocrap	17	90
142	monome-pages	163	256
148	ngamejava	66	21
149	object-procedural-bridge	297	73
152	onslaught	84	26
157	p2ploan	197	192
164	powerjava	28	41
165	powermock	755	689
166	prettyfaces	163	75
168	product-center	283	86
169	project-armageddon	14	31
170	projet-qcm-java	55	104
172	ps3mediaserver	215	769
179	restfb	92	215
180	robust-coupe	69	51
183	scikit	272	515
184	semanticdiscoverytoolkit	1,436	435
185	semweb4j	482	366
186	seoma	307	167
188	simplenamingservice	75	35
189	sjava-logging	21	44
195	squabble	144	51
197	subitizer	16	38
201	tabulasoftmed	323	89
202	tabuvrp–study	36	39
211	usemon	1,090	39

12. ascrblr: a java application programming interface (API) for Audioscrobbler Protocols and Web Services. Audioscrobbler is a service that tracks listening habits.

13. audao: audao is a tool for generating SQL - DDL and DAO libraries. Its input is a configuration file (XML) describing the database entities and relations. The output is a set of SQLs (CREATE/DROP/INSERT) and a java jar library plus Javadoc (optional).

14. bitlyj: a java interface to the bit.ly and j.mp APIs.

15. bluecove: blueCove is a java bluetooth API implementation that interfaces with the Mac OS X, WIDCOMM, BlueSoleil and Microsoft Bluetooth stack in Windows XP SP2. The software was initially built by Intel Research and presently managed by volunteers.

16. castanea: a java project built for exploring the possibilities of java game development.

17. catchnthrow: a java control project for the "catch and throw" process.

18. daedalum: daedalum is an experimental pure java video player framework built on top of the Xuggler ffmpeg wrapper.

19. dbmigrate: this java library aids the management of database upgrades for java applications.

20. echo-nest-java-api: this is a java API and assorted tools and helpers for the Echo Nest API (at `http://developer.echonest.com`>`developer.echonest.com`).

21. fdelimitedtextutilities:a group of java classes to build a delta files based delimited text files

22. fyllgen: a java based tool for collecting and distributing family data.

23. geocoder-java: java API for Google geocoder version 3.

24. google-voice-java: an Unofficial java API for Google Voice.

25. gorobot: java program to play the board game go.

26. gp-net-radius: a java radius protocol stack.

27. guavatools: this software consists of a set of generic java tools that assist with project management when moving from company to company.

28. guitarjava: guitar Hero game on a java Applet.

29. hobbylinkchecker: a java based link checker. This is a specialized link checker built to parse a particular format of webpages. additionally, it will support a basic dfs link checking format that will parse the html and add all the links to a queue. it will then search/verify the links (http, https, or ftp), and return which links failed and which passed.

30. jangod: java Django template engine.

31. jaque: this library provides type safe language integrated query capabilities to Java language.

32. java-chess-web: java-chess-web is an implementation of the game of chess written in Java, with a front end Ajax web application.

33. java-weather-api: very simple API for getting basic weather information from various providers. It is aimed for simplifying retrieval of weather information in applications. It supports three weather providers namely: Google weather service, Yahoo weather service and Weather underground.

34. javacoder:

35. javastepbystep: a java study tool to learn java programming language.

36. jbal: this project aims to develop a java library to access different bibliographic record types with a uniform interface api. The starting point is some code developed for an open-source open public access catalog (opac) engine.

37. jbandwidthlog: the JBandwidth logger is a java implementation of a bandwidth monitor and logger for Windows XP, Vista and 7.

38. jease: jease eases the development of content- database-driven web-applications with java.

39. jeudi-tech-spring: jeudi-tech-spring is a Java web-app demo that uses several modules of the Spring Framework. The purpose of this project is to learn Spring.

40. jiopi: jiopi is the abbreviation of java Interface-Oriented Programming initiative . Its mission is to create open specification for building a java system which is composed of modules.

41. jmemcache: This project is intended to allow easy cache management in local memory for java projects.

42. jnoob: according to the author of this software repository, "JNoob is my personal java repository, can I use google code for this ? ;) Really, since I'm learning java I decided to create a google code repository to keep my files from my home to work and college, so I can happy code them anytime".

43. jothelo: Othelo game developed in java Language.

44. jprg2-assg: this is a java application created to meet the requirements of a JPRG 2 Module, a university project.

45. jroguedps: a java application to provide a World of Warcraft DPS simulation for the Rogue Class.

46. jsbe: a simple java Sub Editor to Resync and edit your .srt files.

47. jtowerdefense: This game is a tower defense done in java. This is the basis for a 2D game engine.

48. jugile-util: java utilities to make coding more convenient.

49. jutf8search: an efficient string matching library in java for UTF-8 encoded text.

50. kryo: kryo is a fast and efficient object graph serialization framework for java. The goals of the project are speed, efficiency, and an easy to use API. The project is effective when objects need to be persisted, whether to a file, database, or over the network.

51. lemyriapode: this software consists of a set of java projects for various educational topics in physics, and maths.

52. migrator-postgresql: software in java to do a migration of data from one postgre DB to another.

53. mobs: this is project for university mobile services subject. Here You can find application, for monitoring information from the GPS or GSM device through RS232 serial port, or log file. It draws provided coordinites as a point with computed bias on specified map. A set of Vilnius maps is also provided.

54. mocrap: java motion capture editing experiment.

55. monome-pages: pages is a monome application that allows the simultaneous execution of multiple other monome applications on any number of devices. There are many built in applications as well, including MIDI interfaces (keyboard, triggers, faders, sequencers), Ableton clip launcher interfaces, and a way to easily script your own programs with Groovy.

56. ngamejava: java library for game creation.

57. object-procedural-bridge: this is a framework for building Java applications that use Oracle PL/SQL.

58. onslaught: java implementation of the Onslaught flash game

59. p2ploan: java application to conduct business loans between users.

60. powerjava: a parser to transform powerjava code into java 1.5 code.

61. powermock: powerMock is a java framework that allows you to unit test code normally regarded as untestable.

62. prettyfaces: prettyFaces is an Open-Source URL-rewriting library with enhanced support for JavaServer Faces – JSF versions 1.1, 1.2 and 2.0 – enabling creation of bookmark-able, pretty URLs. PrettyFaces solves the "RESTful URL" problem, including features such as: seamless integration with faces navigation, managed parameter parsing, page-load actions, dynamic view-id assignment, and configuration-free compatibility with other web frameworks.

63. product-center:

64. project-armageddon: raiden-esque java Game

65. projet-qcm-java: a java tool for quality control management.

66. ps3mediaserver: PS3 media server is a digital living network alliance (DLNA) compliant universal plug and play (UPNP) Media Server for the PS3, written in Java, with the purpose of streaming or transcoding any kind of media files, with minimum configuration. It is also includes robust Mplayer/FFmpeg packages.

67. restfb: a simple java interface for the Facebook Graph and old representational state transfer (REST) API.

68. robust-coupe: robust-coupe aims to provide some features to improve the robust of java application.

69. scikit: java scientific library which facilitates graphics visualization and user interaction. Contains threading support, plotting functionality, data containers, and more.

70. semanticdiscoverytoolkit: java utilities for AI problem solving.

71. semweb4j: semweb4j project provides libraries to do complicated things with Java and RDF. Very little semantic web knowledge is required in order to make use of it.

72. seoma: multiplayer java game.

73. simplenamingservice: a java implementation of the CORBA Naming Service API.

74. sjava-logging: a java library for logging with high performance.

75. squabble: squabble is a content management system (CMS) and chat system written in Java.

76. subitizer: a java program to practice subitizing and/or counting. The Program tracks the users statistics and can be executed as an applet.

77. tabulasoftmed: tabulasoftmed is a java Web Framework, which consists of several sub-projects, such as a server and graphical user interface tools. A preview version of the server is available for Windows and Linux.

78. tabuvrp–study: tabu search for vehicle routing problem. Toy java implementation as case study.

79. usemon: usemon is a monitoring system for usage trends, response time and dependency analysis of plain java applications or big multi-clustered java Enterprise applications running in production.

A.3 Advantages of the vector space model (VSM) compared to latent semantic indexing (LSI) information retrieval (IR) approach

The main disadvantage of LSI (compared to VSM) when applied in the computation of sentence similarity is the lack of potentially important syntactic information. For example, the sentences “The dog chased the man” and “The man chased the dog” are viewed as identical by LSI.

Also, negations and antonyms are not processed by LSI [48, 186]. Research has shown that LSI is less effective at discriminating strongly from weakly-related words because LSI implicitly contains the theory of synonymy.

Lastly, LSI is also not scalable when new documents, not analyzed during model building are parsed using pre-built models, as the concepts or topics in such documents are not captured in the model which has to be re-built.

Appendix B

Appendix B contains the results for Chapter 4.

B.1 Structural and logical coupling interplay outcomes

Table 3 includes results on the intersection between structural and logical class dependencies. The first column includes the project IDs, followed by the names of projects in the second column, the third column contains the number of structural dependencies per project, the fourth column contains the number of logical dependencies per project, the fifth column contains the intersection set (classes linked by both structural and logical dependencies). The sixth column contains the percentage of structural dependencies in the intersection set while the last and seventh column contains the proportion of logical dependencies in the intersection set.

Table 3: Intersection of structural and logical dependencies in the studied 79 OSS projects. (KEY: Str. Dep. = structural dependencies; Log. Dep. = logical dependencies; CSD = co-changed structural dependencies; CLD = coupled logical dependencies)

ID	Project	Str. Dep.	Log. Dep.	Int. Set	CSD (%)	CLD (%)
1	2dtetris	91	166	44	48	27
2	4-connect	18	80	17	94	21
7	ahs-scheduling	65	118	39	60	33
8	aima-java	4,082	190,432	3,200	78	2
10	alexo-chess	655	9,603	499	76	5
11	algmusic	218	3,812	163	75	4

Continued on next page

Table 3 – *Continued from previous page*

ID	Project	Str. Dep.	Log. Dep.	Int. Set	CSD	CLD
12	alleywayreinvented	118	680	118	100	17
13	alto	1,662	78,481	1,567	94	2
14	amock	1,084	2,969	545	50	18
18	apjava	67	196	61	91	31
20	appletbomberman	282	1,255	230	82	18
22	ascrblr	161	1,396	128	80	9
24	audao	317	6,838	198	62	3
26	bitlyj	194	1,036	162	84	16
28	bluecove	753	63,404	607	81	1
30	castanea	157	624	100	64	16
31	catchnthrow	50	164	34	68	21
41	daedalum	252	4,854	249	99	5
45	dbmigrate	13	26	10	77	38
51	echo-nest-java-api	143	1,116	127	89	11
56	fdelimitedtextutilities	31	34	8	26	24
60	fyllgen	674	14,318	656	97	5
64	geocoder-java	120	379	58	48	15
65	google-voice-java	160	724	81	51	11
66	gorobot	2,914	88,731	2,173	75	2
67	gp-net-radius	76	522	61	80	12
68	guavatools	407	6,899	363	89	5
69	guitarjava	309	3,681	248	80	7
71	hobbylinkchecker	476	35,923	473	99	1
79	jangod	802	15,220	697	87	5
81	jaque	368	1,065	205	56	19
84	java-chess-web	659	2,596	337	51	13
86	java-weather-api	52	220	37	71	17
88	javacoder	16	104	16	100	15
92	javastepbystep	259	1,795	109	42	6

Continued on next page

Table 3 – *Continued from previous page*

ID	Project	Str. Dep.	Log. Dep.	Int. Set	CSD	CLD
96	jbal	480	12,986	461	96	4
97	jbandwidthlog	57	468	57	100	12
99	jease	1,365	39,842	861	63	2
103	jeudi-tech-spring	80	310	47	59	15
107	jiopi	73	532	52	71	10
109	jmemcache	24	94	21	88	22
113	jothelo	55	148	42	76	28
112	jnoob	57	417	48	84	12
115	jprg2-assg	83	332	79	95	24
118	jroguedps	673	6,255	532	79	9
119	jsbe	23	70	23	100	33
122	jtowerdefense	231	2,191	210	91	10
123	jugile-util	237	3,088	144	61	5
124	jutf8search	43	152	41	95	27
127	kryo	675	5,372	580	86	11
130	lemyriapode	1,045	10,520	809	77	8
136	migrator-postgresql	78	476	76	97	16
140	mobs	127	672	96	76	14
141	mocrap	47	74	21	45	28
142	monome-pages	835	10,362	727	87	7
148	ngamejava	189	1,196	139	74	12
149	object-procedural-bridge	1,526	27,343	852	56	3
152	onslaught	297	5,739	289	97	5
157	p2ploan	1,185	10,041	750	63	7
164	powerjava	49	150	29	59	19
165	powermock	2,372	105,733	1,828	77	2
166	prettyfaces	519	12,987	500	96	4
168	product-center	1,018	7,220	530	52	7
169	project-armageddon	50	68	30	60	44

Continued on next page

Table 3 – *Continued from previous page*

ID	Project	Str. Dep.	Log. Dep.	Int. Set	CSD	CLD
170	projet-qcm-java	191	868	122	64	14
172	ps3mediaserver	1,177	29,313	983	84	3
179	restfb	407	4,045	303	74	7
180	robust-coupe	367	1,648	182	50	11
183	scikit	1,457	10,958	924	63	8
184	semanticdiscoverytoolkit	6,594	177,962	4,741	72	3
186	seoma	1,341	16,929	1,104	82	7
185	semweb4j	3,954	68,309	2,551	65	4
188	simplenamingservice	274	1,593	205	75	13
189	sjava-logging	53	408	53	100	13
195	squabble	376	4,578	267	71	6
197	subitizer	59	176	47	80	27
201	tabulasoftmed	1,652	58,420	1,373	83	2
202	tabuvrp–study	121	442	72	60	16
211	usemon	4,094	529,590	3,845	94	1

Table 4 contains the results derived from computing the Spearman's rank correlation between strengths of structural and logical coupling of classes in the 79 projects studied. The first column contains the project IDs, followed by project names in the second column, the correlation coefficient in the third column and finally the associated p-values in the fourth and last column.

Table 4: Structural and logical coupling Spearman's rank correlation coefficient outcomes with p-values

ID	Project	Correlation Coefficient	p-value
1	2dtetris	0.06	0.4
2	4-connect	0.01	0.9

Continued on next page

Table 4 – *Continued from previous page*

ID	Project	Correlation Coefficient	p-value
7	ahs-scheduling	0.007	1.0
8	aima-java	NA	NA
10	alexo-chess	NA	NA
11	algmusic	NA	NA
12	alleywayreinvented	NA	NA
13	alto	NA	NA
14	amock	0.02	0.31
18	apjava	0.28	0
20	appletbomberman	0.02	0.54
22	ascrblr	NA	NA
24	audao	NA	NA
26	bitlyj	NA	NA
28	bluecove	NA	NA
30	castanea	0.11	0.003
31	catchnthrow	0.19	0.01
41	daedalum	NA	NA
45	dbmigrate	-0.08	0.7
51	echo-nest-java-api	NA	NA
56	fdelimitedtextutilities	-0.65	0
60	fyllgen	NA	NA
64	geocoder-java	0.06	0.23
65	google-voice-java	0.06	0.07
66	gorobot	NA	NA
67	gp-net-radius	NA	NA
68	guavatools	NA	NA
69	guitarjava	NA	NA
71	hobbylinkchecker	NA	NA
79	jangod	NA	NA
81	jaque	0.01	0.78

Continued on next page

Table 4 – *Continued from previous page*

ID	Project	Correlation Coefficient	p-value
84	java-chess-web	NA	NA
88	javacoder	0.15	0.12
92	javastepbystep	-0.01	0.58
86	java-weather-api	0.08	0.20
96	jbal	NA	NA
97	jbandwidthlog	0.36	0
99	jease	NA	NA
103	jeudi-tech-spring	-0.05	0.32
107	jiopi	NA	NA
109	jmemcache	0.05	0.62
112	jnoob	0.18	< 0.001
113	jothelo	0.26	0.001
115	jprg2-assg	0.23	0
118	jroguedps	NA	NA
119	jsbe	0.28	0.02
122	jtowerdefense	NA	NA
123	jugile-util	NA	NA
124	jutf8search	0.13	0.12
127	kryo	NA	NA
130	lemyriapode	NA	NA
136	migrator-postgresql	0.13	0.006
140	mobs	0.1	0.01
141	mocrap	-0.3	0.01
142	monome-pages	NA	NA
148	ngamejava	NA	NA
149	object-procedural-bridge	NA	NA
152	onslaught	NA	NA
157	p2ploan	NA	NA
164	powerjava	NA	NA

Continued on next page

Table 4 – *Continued from previous page*

ID	Project	Correlation Coefficient	p-value
165	powermock	NA	NA
166	prettyfaces	NA	NA
168	product-center	NA	NA
169	project-armageddon	-0.07	0.51
170	projet-qcm-java	NA	NA
172	ps3mediaserver	NA	NA
179	restfb	NA	NA
180	robust-coupe	NA	NA
183	scikit	NA	NA
184	semanticdiscoverytoolkit	NA	NA
185	semweb4j	NA	NA
186	seoma	NA	NA
188	simplenamingservice	-0.01	0.66
189	sjava-logging	NA	NA
195	squabble	NA	NA
197	subitizer	0.07	0.32
201	tabulasoftmed	NA	NA
202	tabuvrp–study	0.03	0.47
211	usemon	NA	NA

Appendix C

Appendix C contains the results for Chapter 5.

C.1 Semantic and logical coupling interplay outcomes

Table 5 summarises the projects analysed in a pilot study in which we compared the measurement of semantic coupling between classes in two distinct ways: (1) using only their identifiers, and; (2) using the whole corpora.

Table 6 includes results on the intersection between semantic and logical class dependencies. The first column includes the project IDs, followed by the names of projects in the second column, the third column contains the number of semantic dependencies per project, the fourth column contains the number of logical dependencies per project, the fifth column contains the intersection set (classes linked by both semantic and logical dependencies). The sixth column contains the percentage of semantic dependencies in the intersection set while the last and seventh column contains the proportion of logical dependencies in the intersection set.

Table 5: Characteristics of the software systems analyzed for semantic coupling measurement comparison

Project	Classes	Class Pairs	LOC (with comments)	Time to Analyze Corpora (mins)	Time to Analyze Identifiers (mins)	Δ mins
4-connect	10	45	1,160	0.003	0.005	<1%
alexo-chess	119	7,021	22,986	1.01	0.07	143%
alto	315	49,455	101,379	20	1	19%
audao	152	11,476	20,347	1.1	0.1	10%
bitlyj	22	231	1,255	0.002	0.002	0%
bluecove	390	75,855	75,237	18	1	17%
daedalum	68	2,278	10,172	0.2	0.01	19%
dbmigrate	7	21	1,337	0.003	< 0.001	598%
echo-nest-java-api	36	630	6,903	0.1	0.005	19%
fdelimitedtextutilities	11	55	1,769	0.003	0.001	2%
geocoder-java	27	351	1,732	0.006	0.003	1%
google-voice-java	56	1,540	10,078	0.3	0.02	14%
gp-net-radius	25	300	2,469	0.01	0.002	4%
guitarjava	87	21	18,331	0.5	0.03	16%
jangod	127	8,001	10,789	0.4	0.02	19%
java-chess-web	111	6,105	7,983	0.2	0.04	4%
java-weather-api	35	595	2,041	0.01	0.004	2%
jbal	109	5,886	21,285	2	0.04	49%
jbandwidthlog	13	78	2,472	0.01	0.001	9%
jiopi	22	231	2,260	0.003	0.001	2%
jmemcache	14	91	1,035	0.002	0.001	1%
kryo	52	1,326	6,356	0.1	0.01	9%
migrator-postgresql	29	406	2,282	0.01	0.002	4%
monome-pages	158	12,403	64,942	9	0.08	112%
powermock	673	226,128	73,985	21	3	6%
prettyfaces	229	26,106	26,104	2	0.08	24%
projet-qcm-java	53	1,378	4,661	0.04	0.01	3%
ps3mediaserver	189	17,766	39,816	6	0.05	119%
restfb	75	2775	16,041	0.8	0.06	12%
scikit	109	5,886	18,224	1	0.03	32%
semanticdiscoverytoolkit	1,421	1,008,910	268,564	695	7.2	98%
seoma	280	39,060	37,007	2.4	0.3	7%
sjava-logging	19	171	1,514	0.002	0.001	1%
tabuvrp–study	28	378	2,524	0.01	0.002	4%
usemon	1,090	593,505	219,546	980	4	244%

Table 6: Intersection of semantic and logical dependencies in the studied 79 OSS projects. Semantic coupling measured using N-Gram technique (KEY: Sem. Dep. = semantic dependencies; Log. Dep. = logical dependencies; CSD = co-changed semantic dependencies; SLD = semantic logical dependencies)

ID	Project	Sem. Dep.	Log. Dep.	Int. Set	CSD (%)	SLD (%)
1	2dtetris	213	166	166	78	100
2	4-connect	55	80	54	98	68
7	ahs-scheduling	144	118	118	82	100
8	aima-java	190,694	190,432	189,812	100	100
10	alexo-chess	9,759	9,603	9,603	98	100
11	algmusic	3,867	3,812	3,812	99	100
12	alleywayreinvented	668	680	668	100	98
13	alto	77,600	78,481	77,505	100	99
14	amock	3,508	2,969	2,969	85	100
18	apjava	202	196	196	97	100
20	appletbomberman	1,307	1,255	1,255	96	100
22	ascrblr	1,429	1,396	1,396	98	100
24	audao	6,957	6,838	6,838	98	100
26	bitlyj	1,068	1,036	1,036	97	100
28	bluecove	63,358	63,404	63,212	100	100
30	castanea	681	624	624	92	100
31	catchnthrow	180	164	164	91	100
41	daedalum	4,855	4,854	4,852	100	100
45	dbmigrate	29	26	26	90	100
51	echo-nest-java-api	1,132	1,116	1,116	99	100
56	fdelimitedtextutilities	57	34	34	60	100
60	fyllgen	14,316	14,318	14,298	100	100
64	geocoder-java	441	379	379	86	100
65	google-voice-java	767	724	694	90	96
66	gorobot	89,362	88,731	88,627	99	100
67	gp-net-radius	537	522	522	97	100
68	guavatools	6,923	6,899	6,879	99	100

Continued on next page

Table 6 – *Continued from previous page*

ID	Project	Sem. Dep.	Log. Dep.	Int. Set	CSD (%)	SLD (%)
69	guitarjava	3,412	3,681	3,351	98	91
71	hobbylinkchecker	35,890	35,923	35,887	100	100
79	jangod	15,126	15,220	15,030	99	99
81	jaque	1,228	1,065	1,065	87	100
84	java-chess-web	2,902	2,596	2,590	89	100
86	java-weather-api	231	220	216	94	98
88	javacoder	104	104	104	100	100
92	javastepbystep	1,945	1,795	1,795	92	100
96	jbal	12,903	12,986	12,884	100	99
97	jbandwidthlog	468	468	468	100	100
99	jease	40,346	39,842	39,842	99	100
103	jeudi-tech-spring	343	310	310	90	100
107	jiopi	553	532	532	96	100
109	jmemcache	97	94	94	97	100
112	jnoob	426	417	417	98	100
113	jothelo	137	148	124	91	84
115	jprg2-assg	336	332	332	99	100
118	jroguedps	6,394	6,255	6,253	98	100
119	jsbe	70	70	70	100	100
122	jtowerdefense	2,212	2,191	2,191	99	100
123	jugile-util	3,175	3,088	3,082	97	100
124	jutf8search	152	152	150	99	99
127	kryo	5,465	5,372	5,370	98	100
130	lemyriapode	10,732	10,520	10,496	98	100
136	migrator-postgresql	478	476	476	100	100
140	mobs	703	672	672	96	100
141	mocrap	100	74	74	74	100
142	monome-pages	10,462	10,362	10,354	99	100
148	ngamejava	1,246	1,196	1,196	96	100

Continued on next page

Table 6 – *Continued from previous page*

ID	Project	Sem. Dep.	Log. Dep.	Int. Set	CSD (%)	SLD (%)
149	object-procedural-bridge	27,983	27,343	27,309	98	100
152	onslaught	5,747	5,739	5,739	100	100
157	p2ploan	10,476	10,041	10,041	96	100
164	powerjava	168	150	148	88	99
165	powermock	105,828	105,733	105,291	99	100
166	prettyfaces	12,968	12,987	12,949	100	100
168	product-center	7,708	7,220	7,220	94	100
169	project-armageddon	88	68	68	77	100
170	projet-qcm-java	937	868	868	93	100
172	ps3mediaserver	29,497	29,313	29,303	99	100
179	restfb	4,139	4,045	4,035	97	100
180	robust-coupe	1,833	1648	1,648	90	100
183	scikit	11,489	10,958	10,956	95	100
184	semanticdiscoverytoolkit	179,777	177,962	177,928	99	100
185	semweb4j	69,401	68,309	68,009	98	100
186	seoma	17,166	16,929	16,929	99	100
188	simplenamingservice	1,662	1,593	1,593	96	100
189	sjava-logging	408	408	408	100	100
195	squabble	4,687	4,578	4,578	98	100
197	subitizer	188	176	176	94	100
201	tabulasoftmed	58,497	58,420	58,218	100	100
202	tabuvrp–study	491	442	442	90	100
211	usemon	529,180	529,590	528,932	100	100

Table 7: Spearman's rank correlation results for the linear relationship between logical and semantic coupling strengths in the overall sample of OSS projects

ID	Project	Correlation Coefficient	p-value
1	2dtetris	0.2	0
2	4-connect	-0.3	0.02
7	ahs-scheduling	0.01	0.9
8	aima-java	NA	NA
10	alexo-chess	NA	NA
11	algmusic	NA	NA
12	alleywayreinvented	NA	NA
13	alto	NA	NA
14	amock	0.04	0.01
18	apjava	0.3	8.8-e06
20	appletbomberman	0.1	0.01
22	ascrblr	NA	NA
24	audao	NA	NA
26	bitlyj	NA	NA
28	bluecove	NA	NA
30	castanea	0.1	0.03
31	catchnthrow	0.03	0.8
41	daedalum	NA	NA
45	dbmigrate	0.1	1
51	echo-nest-java-api	NA	NA
56	fdelimitedtextutilities	-0.04	0.7
60	fyllgen	NA	NA
64	geocoder-java	0.1	0.2
65	google-voice-java	-0.02	0.5
66	gorobot	NA	NA
67	gp-net-radius	NA	NA
68	guavatools	NA	NA
69	guitarjava	NA	NA

Continued on next page

Table 7 – *Continued from previous page*

ID	Project	Correlation Coefficient	p-value
71	hobbylinkchecker	NA	NA
79	jangod	NA	NA
81	jaque	0.1	0.1
84	java-chess-web	NA	NA
86	java-weather-api	0	1
88	javacoder	-0.1	0.3
92	javastepbystep	0.04	0.1
96	jbal	NA	NA
97	jbandwidthlog	-0.2	7.2e-06
99	jease	NA	NA
103	jeudi-tech-spring	0.3	3.5e-07
107	jiopi	NA	NA
109	jmemcache	0.1	0.2
112	jnoob	0.1	0.3
113	jothelo	0.3	0
115	jprg2-assg	0.02	0.6
118	jroguedps	NA	NA
119	jsbe	0.2	0.04
122	jtowerdefense	NA	NA
123	jugile-util	NA	NA
124	jutf8search	0.4	2.5e-06
127	kryo	NA	NA
130	lemyriapode	NA	NA
136	migrator-postgresql	0.2	4.8-e06
140	mobs	0.02	0.8
141	mocrap	0.03	0.7
142	monome-pages	NA	NA
148	ngamejava	NA	NA
149	object-procedural-bridge	NA	NA

Continued on next page

Table 7 – *Continued from previous page*

ID	Project	Correlation Coefficient	p-value
152	onslaught	NA	NA
157	p2ploan	NA	NA
164	powerjava	0.02	0.8
165	powermock	NA	NA
166	prettyfaces	NA	NA
168	product-center	NA	NA
169	project-armageddon	0.1	0.9
170	projet-qcm-java	-0.1	0.5
172	ps3mediaserver	NA	NA
179	restfb	NA	NA
180	robust-coupe	NA	NA
183	scikit	NA	NA
184	semanticdiscoverytoolkit	NA	NA
185	semweb4j	NA	NA
186	seoma	NA	NA
188	simplenamingservice	0.1	0.01
189	sjava-logging	0.1	0.1
195	squabble	NA	NA
197	subitizer	0.1	0.1
201	tabulasoftmed	NA	NA
202	tabuvrp–study	0.1	0
211	usemon	NA	NA

Appendix D

Appendix D contains the results for Chapter 6.

D.1 Structural and semantic coupling interplay outcomes

Table 8 consists of the results derived from computing the Fisher's independence tests to determine the presence or absence of a significant association between structural and semantic class dependencies. The first column contains the project names, and the second column contains the number of class pairs linked both structurally and semantically. The third column contains the number of class pairs linked structurally but not semantically while the fourth column contains the number of class pairs linked semantically but not structurally.

The fifth column in Table 8 contains the number of class pairs that are not linked either structurally or semantically. The last column contains the resulting p-values used to reject or fail to reject the null hypothesis which asserts that there is not a significant association between structural and semantic class dependencies.

Table 8: Structural and semantic coupling 2x2 contingency table outcomes with Fisher's independence test p-values – overall sample of OSS projects

ID	project	E	H	W	x	p-value
1	2dtetris	16	64	5	0	<0.001

Continued on next page

Table 8 – *Continued from previous page*

ID	project	E	H	W	x	p-value
2	4-connect	0	18	0	1	1
7	ahs-scheduling	6	52	1	3	0.4
8	aima-java	547	2,875	2,627	193	0
10	alexo-chess	111	527	266	1	5.5e-137
11	algmusic	37	179	166	29	1.2e-46
12	alleywayreinvented	10	101	12	16	8.2e-05
13	alto	258	1,328	1,809	28	0
14	amock	119	931	72	2	1.6e-58
18	apjava	8	35	8	4	0.00266
20	appletbomberman	40	222	77	1	4.4e-45
22	ascrblr	33	126	59	6	0
24	audao	88	220	832	38	9.7e-123
26	bitlyj	45	141	76	3	2.5e-30
28	bluecove	142	559	1,338	423	5.8e-146
30	castanea	18	122	22	6	1.04e-11
31	catchnthrow	5	42	10	0	7.0e-08
41	daedalum	35	211	98	78	1.31e-19
45	dbmigrate	9	4	2	0	1
51	echo-nest-java-api	10	103	15	21	2.4e-05
56	fdelimitedtextutilities	3	28	1	0	0.13
60	fyllgen	65	578	229	15	3.3e-130
64	geocoder-java	39	64	43	0	2.1e-14
65	google-voice-java	31	88	59	1	6.5e-23
66	gorobot	375	2,410	1,822	69	0
67	gp-net-radius	39	35	65	6	1.3e-07
68	guavatools	128	274	668	75	9.3e-93
69	guitarjava	55	212	65	18	1.9e-21
71	hobbylinkchecker	192	275	772	91	7.6e-78
79	jangod	89	692	575	24	2.1e-252

Continued on next page

Table 8 – *Continued from previous page*

ID	project	E	H	W	x	p-value
81	jaque	74	262	53	2	1.4e-27
84	java-chess-web	121	499	251	0	8.4e-126
86	java-weather-api	24	27	19	0	1.5e-05
88	javacoder	2	14	11	5	0.003
92	javastepbystep	39	194	116	44	2.2e-29
96	jbal	45	431	174	49	1.1e-74
97	jbandwidthlog	14	35	18	4	6.3e-05
99	jease	240	1,117	1,215	78	0
103	jeudi-tech-spring	17	63	20	2	2.8e-09
107	jiopi	18	54	17	4	5.6e-06
109	jmemcache	6	17	3	1	0.1
112	jnoob	18	34	32	12	<0.001
113	jothelo	0	44	2	3	0.01
115	jprg2-assg	19	54	20	0	6.5e-10
118	jroguedps	190	418	473	10	1.4e-133
119	jsbe	1	16	1	2	0.1
122	jtowerdefense	41	180	76	12	3.2e-29
123	jugile-util	50	157	207	21	0
124	jutf8search	6	34	5	1	0.002
127	kryo	137	494	449	7	5.2e-168
130	lemyriapode	146	815	550	213	1.3e-132
136	migrator-postgresql	7	59	19	7	9.4e-09
140	mobs	26	99	20	5	2.8e-08
141	mocrap	6	41	5	2	0.003
142	monome-pages	167	582	555	12	4.8e-197
148	ngamejava	39	127	71	3	2.4e-28
149	object-procedural-bridge	289	1,214	1,216	5	0
152	onslaught	57	219	144	29	1.2e-40
157	p2ploan	264	869	530	8	2.4e-218

Continued on next page

Table 8 – *Continued from previous page*

ID	project	E	H	W	x	p-value
164	powerjava	3	45	4	4	0.01
165	powermock	428	1,883	1,487	91	0
166	prettyfaces	103	403	329	7	7.7e-129
168	product-center	389	626	840	27	1.1e-183
169	project-armageddon	4	40	0	0	1
170	projet-qcm-java	35	107	69	0	1.2-29
172	ps3mediaserver	139	925	390	88	3.3e-154
179	restfb	65	319	127	18	1.4-52
180	robust-coupe	31	290	76	0	0
183	scikit	119	1,289	327	30	4.9e-213
184	semanticdiscoverytoolkit	1212	4,587	3,538	137	0
185	semweb4j	331	2,759	1,030	95	0
186	seoma	121	912	283	67	2.6e-128
188	simplenamingservice	50	181	58	1	0
189	sjava-logging	6	44	16	6	6.94e-07
195	squabble	125	178	348	14	4.3e-60
197	subitizer	12	34	5	0	0.003
201	tabulasoftmed	175	1,270	787	58	0
202	tabuvrp–study	25	73	18	0	1.1e-09

Similarly to Table 8, Table 9 consists of the results derived from computing the Chi-square independence tests to determine the presence or absence of a significant association between structural and semantic class dependencies. The first column contains the project names, and the second column contains the number of class pairs linked both structurally and semantically. The third column contains the number of class pairs linked structurally but not semantically while the fourth column contains the number of class pairs linked semantically but not structurally.

The fifth column in Table 9 contains the number of class pairs that are not linked either structurally or semantically. The last column contains the resulting p-values used to reject or fail to reject the null hypothesis which asserts that there is not a significant association between structural and semantic class dependencies.

Table 9: Structural and semantic coupling 2x2 Contingency table outcomes with Chi-square independence test p-values – overall sample of OSS projects

ID	project	E	H	W	x	p-value
1	2dtetris	16	64	5	0	<0.001
2	4-connect	0	18	0	1	NaN
7	ahs-scheduling	6	52	1	3	0.9
8	aima-java	547	2,875	2,627	193	0
10	alexo-chess	111	527	266	1	3.7e-115
11	algmusic	37	179	166	29	1.5e-42
12	alleywayreinvented	10	101	12	16	4.2e-05
13	alto	258	1,328	1,809	28	0
14	amock	119	931	72	2	0
18	apjava	8	35	8	4	0.004
20	appletbomberman	40	222	77	1	2.0e-41
22	ascrblr	33	126	59	6	1.8e-21
24	audao	88	220	832	38	3.1e-131
26	bitlyj	45	141	76	3	2.2e-26
28	bluecove	142	559	1,338	423	0
30	castanea	18	122	22	6	5.6e-13
31	catchnthrow	5	42	10	0	5.6e-08
41	daedalum	35	211	98	78	4.2e-19
45	dbmigrate	9	4	2	0	1
51	echo-nest-java-api	10	103	15	21	1.5e-05
56	fdelimitedtextutilities	3	28	1	0	0.24931
60	fyllgen	65	578	229	15	6.2e-123
64	geocoder-java	39	64	43	0	1.9e-11
65	google-voice-java	31	88	59	1	2.91e-19

Continued on next page

Table 9 – *Continued from previous page*

ID	project	E	H	W	x	p-value
66	gorobot	375	2,410	1,822	69	0
67	gp-net-radius	39	35	65	6	5.5e-07
68	guavatools	128	274	668	75	1.1e-91
69	guitarjava	55	212	65	18	1.4e-21
71	hobbylinkchecker	192	275	772	91	1.1e-78
79	jangod	89	692	575	24	1.3e-212
81	jaque	74	262	53	2	5.4e-27
84	java-chess-web	121	499	251	0	3.7e-104
86	java-weather-api	24	27	19	0	<0.001
88	javacoder	2	14	11	5	0.004
92	javastepbystep	39	194	116	44	3.5e-28
96	jbal	45	431	174	49	1.8e-73
97	jbandwidthlog	14	35	18	4	9.2e-05
99	jease	240	1,117	1,215	78	0
103	jeudi-tech-spring	17	63	20	2	8.0e-09
107	jiopi	18	54	17	4	1.1e-05
109	jmemcache	6	17	3	1	0.2
112	jnoob	18	34	32	12	<0.001
113	jothelo	0	44	2	3	0.002
115	jprg2-assg	19	54	20	0	1.3e-08
118	jroguedps	190	418	473	10	1.4e-110
119	jsbe	1	16	1	2	0.7
122	jtowerdefense	41	180	76	12	5.9e-28
123	jugile-util	50	157	207	21	1.2e-44
124	jutf8search	6	34	5	1	0.002
127	kryo	137	494	449	7	7.8e-138
130	lemyriapode	146	815	550	213	7.3e-126
136	migrator-postgresql	7	59	19	7	9.8e-09
140	mobs	26	99	20	5	1.9e-08

Continued on next page

Table 9 – *Continued from previous page*

ID	project	E	H	W	x	p-value
141	mocrap	6	41	5	2	0.002
142	monome-pages	167	582	555	12	2.8e-163
148	ngamejava	39	127	71	3	1.0e-24
149	object-procedural-bridge	289	1,214	1,216	5	0
152	onslaught	57	219	144	29	5.7e-38
157	p2ploan	264	869	530	8	2.6e-181
164	powerjava	3	45	4	4	0.004
165	powermock	428	1,883	1,487	91	0
166	prettyfaces	103	403	329	7	4.5e-107
168	product-center	389	626	840	27	2.3e-155
169	project-armageddon	4	40	0	0	NaN
170	projet-qcm-java	35	107	69	0	4.3e-24
172	ps3mediaserver	139	925	390	88	8.2e-151
179	restfb	65	319	127	18	1.1e-50
180	robust-coupe	31	290	76	0	2.4e-56
183	scikit	119	1,289	327	30	9.1e-228
184	semanticdiscoverytoolkit	1,212	4,587	3,538	137	0
185	semweb4j	331	2,759	1,030	95	0
186	seoma	121	912	283	67	9.8e-133
188	simplenamingservice	50	181	58	1	8.2e-27
189	sjava-logging	6	44	16	6	1.1e-06
195	squabble	125	178	348	14	5.8e-54
197	subitizer	12	34	5	0	0.005
201	tabulasoftmed	175	1,270	787	58	1.1e-313
202	tabuvrp–study	25	73	18	0	9.0e-09

Table 10 contains the results derived from computing the Spearman's rank correlation between strengths of structural and semantic coupling of classes in the 79 projects studied. The first column contains the project names followed by the correlation coefficient in the second column and finally the associated p-values in the third and last column.

Table 10: Spearman's rank correlation results for the linear relationship between structural and semantic coupling strengths in the overall sample of OSS projects

ID	Project	Correlation Coefficient	p-value
1	2dtetris	0.1	0.6
2	aima-java	0.2	1.2e-05
7	alexo-chess	-0.4	2.6e-05
11	algmusic	-0.2	0.4
12	alleywayreinvented	-0.1	1
13	alto	0.1	0.1
14	amock	-0.02	1
18	apjava	-0.4	0.3
20	appletbomberman	-0.04	1
22	ascrblr	0.03	1
24	audao	-0.03	1
26	bitlyj	-0.3	0.1
28	bluecove	0.1	0.4
30	castanea	-0.2	0.3
31	catchnthrow	0.2	1
41	daedalum	0.1	0.6
45	dbmigrate	-0.1	1
51	echo-nest-java-api	-0.3	0.4
60	fyllgen	-0.1	1
64	geocoder-java	-0.2	0.24
65	google-voice-java	0.3	0.1
66	gorobot	0.002	1
67	gp-net-radius	0.2	0.4
68	guavatools	0.3	0.004

Continued on next page

Table 10 – *Continued from previous page*

ID	Project	Correlation Coefficient	p-value
69	guitarjava	-0.01	1
71	hobbylinkchecker	0.1	0.3
79	jangod	0.2	0.13
81	jaque	-0.1	1
84	java-chess-web	0.1	0.3
86	java-weather-api	-0.1	1
92	javastepbystep	-0.42	0.01
96	jbal	-0.2	0.2
97	jbandwidthlog	0.1	1
99	jease	0.04	1
103	jeudi-tech-spring	-0.04	1
107jiopi	-0.03	1	
109	jmemcache	1	0.3
112	jnoob	-0.2	0.4
115	jprg2-assg	0.4	0.1
118	jroguedps	0.03	0.7
122	jtowerdefense	-0.04	0.8
123	jugile-util	-0.1	0.6
127	kryo	-0.02	0.8
130	lemyriapode	0.02	0.8
136	migrator-postgresql	0.01	1
140	mobs	0.2	0.4
142	monome-pages	0.1	0.1
148	ngamejava	0.3	0.1
149	object-procedural-bridge	0.2	0.003
152	onslaught	-0.3	0.02
157	p2ploan	-0.2	9.6e-05
164	powermock	0.1	0.001
166	prettyfaces	0.1	0.5

Continued on next page

Table 10 – *Continued from previous page*

ID	Project	Correlation Coefficient	p-value
168	product-center	0.04	0.4
169	project-armageddon	0.1	0.9
170	projet-qcm-java	-0.4	0.01
172	ps3mediaserver	-0.02	0.8
179	restfb	0.1	0.6
180	robust-coupe	1	1.6e-07
183	scikit	0.2	0.1
184	semanticdiscoverytoolkit	0.1	0.01
185	semweb4j	-0.03	0.5
186	seoma	-0.1	0.5
188	simplenamingservice	-0.4	0.002
189	sjava-logging	0.1	0.8
195	squabble	-0.1	0.6
197	subitizer	0.1	0.7
201	tabulasoftmed	0.04	0.6
202	tabuvrp–study	-0.3	0.1