Understanding the Interplay between the Logical and Structural Coupling of Software Classes

Nemitari Ajienka[1], Andrea Capiluppi[1]

[a]*Brunel University London*
Kingston Lane, Uxbridge
Middlesex, UB8 3PH

Abstract

During the lifetime of object-Oriented (OO) software systems, new classes are added to increase functionality, also increasing the inter-dependencies between classes. Logical coupling depicts the change dependencies between classes, while structural coupling measures source code dependencies induced via the system architecture. The relationship or dependency between logical and structural coupling have been debated in the past, but no large study has confirmed yet their interplay.

In this study, we have analysed 79 open-source software projects of different sizes to investigate the interplay between the two types of coupling. First, we quantified the overlapping or intersection of structural and logical class dependencies. Second, we statistically computed the correlation between the strengths of logical and structural dependencies. Third, we propose a simple technique to determine the *stability* of OO software systems, by clustering the pairs of classes as "stable" or "unstable", based on their co-change pattern.

The results from our statistical analysis show that although there is no strong evidence of a linear correlation between the strengths of the coupling types, there is substantial evidence to conclude that structurally coupled class pairs usually include logical dependencies. However, not all co-changed class pairs are also linked by structural dependencies. Finally, we identified that only a low proportion of structural coupling shows excessive instability in the studied OSS projects.

Email addresses: nemitari.ajienka@brunel.ac.uk (Nemitari Ajienka), andrea.capiluppi@brunel.ac.uk (Andrea Capiluppi)

Keywords: object-oriented (OO), open-source software (OSS), references, structural coupling, co-changed structural dependencies (CSD), coupled logical dependencies (CLD)

1. Introduction

Various software dependency measures have been proposed over the years. *Logical coupling* is a measure of the degree to which two or more classes change together or co-evolve, based on the historical data of modifications; while *structural coupling* is a measure of the structural or source code dependencies between software classes. For example, the number of method calls between object-oriented (OO) software classes.

Establishing that two software entities *co-evolve* (i.e., they are *logically coupled*) means that developers consider them as logically related: for example, a change in one entity causes a change to be made to another entity. This is also known as the cause → effect rule.

On the other hand, *structural coupling* is the degree of interdependence between software modules, and it indicates how closely connected two modules are at the source code level. Henderson-Sellers *et al.* [1] state that strong coupling complicates a system since a module is harder to understand, change, or correct by itself, if it is highly interrelated with other modules. "*Software complexity can be reduced by designing systems with the weakest possible coupling between modules*" [1] because "*every time a supplier class changes, its clients are also likely to change*" [2].

In earlier studies, co-evolution of OO software classes has been studied in relation to structural coupling [2, 3, 4, 5] and software quality [6, 7]. Some of these studies showed that most of the structurally coupled related entities in software projects do not co-evolve, and the other way round [2, 4, 5].

Figure 1 illustrates what has been proposed in the past, and for a smaller subset of classes: analysing the direction of the relationship between co-evolution and structural coupling for 12 Linux kernel modules [3], Yu identified a linear and directional relationship between the co-evolution and structural coupling. According to that work, structural coupling does not bring about independent evolution: if software classes are evolved independently, there will be no correlation between structural coupling and co-evolution data. In addition, according to Oliva and Gerosa [5], controlling coupling levels in practice is still challenging. One of the reasons is that the extent to which changes propagate via structural dependencies is still not clear.

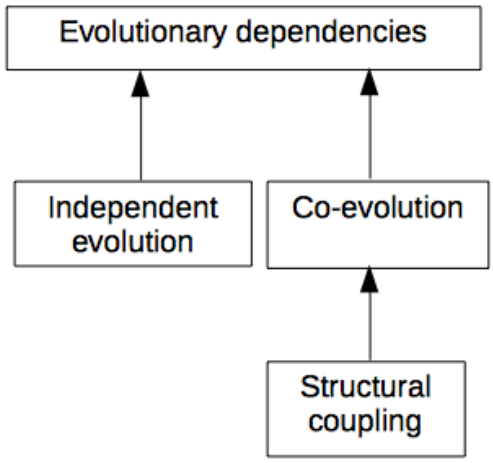

Figure 1: The relationships among evolutionary dependencies, structural coupling and co-evolution [3] of Linux Kernel Modules.

In this context and state of knowledge, this paper analyses a sample of 79 OSS projects (written in Java) in order to add evidence to the discussion on the causes of co-evolution of classes with a large sample of a variety of software projects and to work on the gaps identified in previous research [3, 2].

This work is articulated as follows: in Section 2 we briefly explain the types of software dependencies (coupling) under study. For the sake of replicability, in Section 3 we describe the steps taken to carry out this study, with a working example using a software project. Sections 4 and 5 highlight the findings of our study, followed by a discussion on the importance of these findings. In Section 6 we summarise the related work, and put ours into context. Section 7 highlights the threats to validity and finally, our conclusions and areas for further research are presented in Section 8.

2. Object-Oriented Software Dependencies

A dependency is a semantic relationship that indicates that a client element may be affected by changes performed in a supplier element [2]. In the next Subsections, we introduce structural and logical dependencies and discuss how they can be operationalised in the context of OO programming.

2.1. Logical Coupling

According to Wiese *et al.*, "*change coupling is a phenomenon associated with recurrent co-changes found in the software evolution or change history*" [8]. Therefore, the logical coupling of any two classes is based on their evolution history, and is a measure of the observation that the two classes always co-evolve or change together[9, 10, 11, 12]. They are commonly treated as association rules [6], which means that when X_1 is changed, X_2 is also changed [2]. Furthermore, X1 and X2 are called the antecedent (i.e., left-hand-side, LHS) and the consequent (i.e., right-hand-side, RHS) of the rule, respectively. For example, the rule {A, B}→ C found in the sales data of a supermarket indicates that a customer who buys A and B together, is also likely to buy C [2].

Two classes change at the same time when changes in one class A are made in response to a change in another class B. Kagdi *et al.* [13] state that logical coupling captures the extent to which software artifacts co-evolve and this information is derived by analysing patterns, relationships and relevant information of source code changes mined from multiple versions (of software systems) in software repositories (e.g., Subversion and Bugzilla).

According to Lanza *et al.* [14] it is useful to study logical coupling because it can reveal dependencies that are not revealed by analyzing only the source code [3]. This sort of dependencies are the most troublesome and are prone to represent sources of bugs in software projects. Zimmermann *et al.* [15] represents logical dependency using two metrics: support and confidence.

Operationalisation. *Confidence* and *support* are two well-known metrics used in association rule learning: the *support* value counts the number of revisions where two software artifacts (i.e., classes) were changed together, in other words the probability of finding both the antecedent and consequent in the set of revisions. For example, in Figure 2, class A was modified in 3 transactions (where 3 is the *"Transaction Count"* [3]). Out of these 3 transactions, 2 also included changes to the class C. Therefore, the support for the logical dependency $A \rightarrow C$ will be 2. By its own nature, support is a symmetric metric, so the $A \rightarrow C$ dependency also implies $A \leftarrow C$.

In this paper, the degree or strength of the logical dependency between classes is evaluated using the **confidence** metric. By doing so, we evaluated the *significance* of the association rules between classes [2], and across the lifespan of a software project (i.e., taking all versions of the software system into consideration).

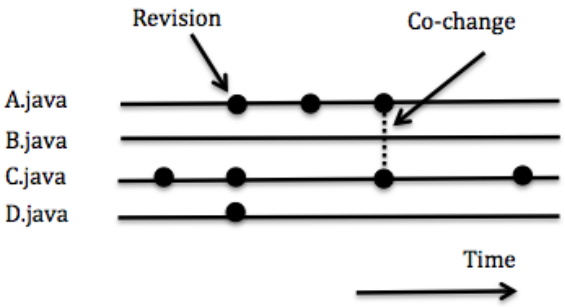

Figure 2: Association rule example for confidence and support metrics

As per its definition, the *confidence*[1] value of a dependency link normalizes the support value by the total number of changes of the causal class, or the antecedent of the association rule. Numerically, it is the ratio of the support count to transaction count: from Figure 2, the confidence value for the association rule $A \rightarrow C$ (which states that C depends on A) will have a high confidence value of $2/3 = 0.67$. In contrast, the rule $C \rightarrow A$ (which states that A depends on C) has a lower confidence value of $2/4 = 0.5$. In other words, the confidence is directional, and determines the strength of the consequence of a given (directional) logical dependency.

Finally, logical coupling is directional, thus $A \rightarrow C$ (changes made to class A resulted in changes in C) and $C \rightarrow A$ (changes in C caused changes in A) will have different meanings. As a result, the confidence for these two cause $\rightarrow$ effect rules can be different.

2.2. Structural Coupling

According to Yu [3], structural coupling is also directional. Geipel and Schweitzer [4] state that there is a directed dependency between two classes A and B if A depends on B in such a way that A is not operational without module B. In the case of Java, this means that A would not compile 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.

[1]Also called the support ratio [3]. In this study we only adopt the confidence metric which is a measure of the degree to which a change in one class also leads to a change in another class

If A depends on B, changes made to B can lead to changes to A, but not the other way round [4]. Therefore, we need to explicitly define the direction of the dependency relationship between these two classes. We adapt Yu's [3] representation of directional coupling: a single directional solid arrow from class A to class B denotes that class B is directionally coupled to class A. This is depicted as **A→B**. We remark that the relation "class B is directionally coupled to class A" is denoted by an arrow from class A to class B. This is because B is dependent on A; a change to class A can affect class B.

Coupling is derived from the number of referring variables and functions of other modules. There are several types of relationships among source code entities (e.g. method calls, class access, or class inheritance). The constructs of most programming languages such as C, C++, and Java can induce such type of relationships [16]. 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 dependency between two classes. These static structural code dependencies are most frequently used when analyzing or leveraging coupling [17].

Operationalisation. In this study, the strength of the structural coupling of classes is measured by the **number of references** from the *caller* class to the *called* class. Oliva and Gerosa measured structural coupling using the Message Passing Coupling (MPC) metric which is the number of external operation calls, i.e. the number of calls from methods of a class to operations of other classes. Yu [3] 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 in component C1 to the use of the function in component C2*" [3]). Accordingly, the strength of the structural coupling from the *caller* C2 to the *called* class C1 in Figure 3 is 4 (2 for function call func(int), 2 for global variable gv) [3].

3. Research Methodology

In this section, we present the research goals and questions to be answered (3.1), we describe the inclusion or selection criteria for the case studies (3.2), data collection (3.3). Additionally, we outline the steps performed in the methodology with the use of worked examples: identifying the class coupling types (3.4); evaluating their intersection (3.5); performing the statistical tests (3.6); and measuring the structural stability of structural links between classes (3.7).

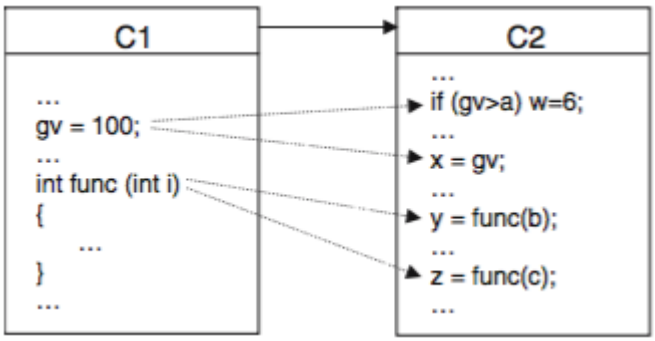

Figure 3: Structural Dependency Path Between Two OO Software Classes – *Caller* (C2) using a function and variable defined in *Called* (C1) [3]

3.1. Research Goals and Questions

This work is based on the three **goals** with related **motivations** presented in Table 1:

G1: to investigate the **interplay** between logical and structural coupling by identifying the proportion of established logical dependencies that involve structurally related elements and the actual proportion of structural dependencies that effectively lead to logical dependencies [2];

G2: to investigate whether the evolutionary coupling **strength** between classes and their structural coupling strength co-vary; and

G3: to cluster the pairs of classes as "stable" or "unstable", based on their frequency of co-change. How the "stability" of a pair of classes was evaluated is discussed in Section 3.7.

Research **questions** were derived from each goal, and testable **hypotheses** formulated, as summarised in Table 1.

With respect to related work, our study is the largest attempt so far (in terms of projects examined) to evaluate the relationship between structural and logical coupling. Also, it is the first work that examines all the revisions of the sampled projects, instead of only one snapshot of their evolution (typically, the last one [4]).

Establishing whether there is an interplay between logical and structural coupling has several **applications** in software engineering, including:

A1 Prediction of software changes: Geipel and Schweitzer [4] state that the question about the causes of change propagation has been overlooked by many researchers in favor of a predictive approach. As such,

Table 1: Research Goals and Questions

Goals	Research Questions	Motivation	Null Hypothesis H_0
G1	[Q1] *Is there a directional relationship between structural and logical coupling?*	While it has been identified that structural coupling leads to logical coupling on a small sample of artifacts [3], others have identified that very often most of the logical dependencies are not caused by structural coupling and vice versa [2, 5, 4]. These varying results need verification on a distinct sample of projects of different sizes.	
G2	[Q2] *Is there a linear relationship between logical and structural coupling?*	The strength of the generalizeability of Yu's study [3] needs further improvement with a larger sample. Furthermore, the chosen $\alpha = 0.1$ might have resulted in a type I error – mistakenly rejecting a null hypothesis.	No linear relationship between the strengths of logical and structural class dependencies in OO software.
G3	[Q3] *What is the proportion of stable pairs of classes in a software system?*	This research goal is exploratory. Taking cues from the structural engineering discipline, the assertion is that as with the renovation of building structures, where maintenance happens for a period of time [18, 19], adding a stable coupling link between two classes should ideally only require little or no co-changes in the first half of their coupling life-cycle.	

these causes are implicitly contained in a prediction function or as input to a machine learning algorithm as done in prior studies on software change prediction [6, 20, 21, 22, 23]. A strong relationship between co-evolution of classes and structural coupling provides statistical support for these models and predictions, thus helping to achieve more focused software maintenance.

A2 Co-change inferred by structural coupling: understanding the influence of structural coupling on co-change can also help in predicting the co-change of software classes based on coupling data, i.e., which classes are likely to be changed based on the internal structure of a software system.

A3 Focusing maintenance effort: If a majority of the classes in an object-oriented software system are not structurally linked but co-change frequently, this will be an indication of the need to investigate other implicit [24] forms of coupling (e.g., semantic or conceptual coupling) [2, 25] which may be propagating ripple effects of changes across classes to reduce maintenance efforts.

A4 Focusing testing effort: the relationship between structural and logical coupling would also help in software testing. When changes are made to one class, other classes with strong co-change or structural coupling to that class should also be tested. This is to ensure that the changes in one class do not introduce regression faults in other classes.

3.2. Case Study

The selection criteria for our sample of projects were based on the following considerations:

1. Open source projects which (i) provide public access to source code and (ii) use a version control system that allows us to extract the historical information [26];
2. Implemented in Java to allow the extraction of the structural coupling between classes, since structural coupling varies between languages [2];
3. Randomly sampled projects;
4. Multiple revisions/commits ($>$ 20 revisions in order to exclude trivial projects), and a relatively long history log. Prior research [27] 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 have 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;

5. A large group of users.

3.3. Empirical Data collection

In the next Subsections, we present how and what kind of data we collected from the repositories of the studied sample of OO software projects.

3.3.1. Selection of a sample of OSS projects

Leveraging the FlossMole project, we used its latest available data dump to determine the population of GoogleCode: a total of 2,593,222 projects are listed in the November 2012 dump.[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: using a 95% confidence level, and a 5% confidence interval, a random sample of 380 IDs were extracted, and linked to the Java projects' names.

3.3.2. Storage of projects metadata and revisions

The first phase of this activity was centered on obtaining the metadata (e.g, name of developers, date and time of changes, etc.) of each project in the sample. The repository of each project was downloaded and stored, with its metadata, using the CVSAnalY set of tools.[3] The metadata allowed us to obtain the list of revisions for each class, and for the whole project, as well as the development and revision logs [28]. Table 2 summarises the sample in terms of number of stored classes and number of revisions per project (Q1 and Q3 represent the first and third quartiles of the distribution of values, respectively).

The second phase was to get all the revisions of each project, from this we could identify the trivial projects (with < 20 revisions) and exclude these from the study. As a result, we ended up with 79 non-trivial Java open-source software projects. Since we also want to calculate the structural and logical coupling between Java classes, in this study, we have excluded revisions without files with the .java extension as our focus is on classes written

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

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

in Java. We have also filtered out revisions with over 10 files [13] to reduce noise in logical coupling measurement and mitigate the influence of factors such as updating licence information in all classes which are not related to changes in source code. Filtering non-structural change couplings reduces the amount of misleading change couplings and, thus, reduces the effort to investigate all change couplings [29].

Table 2: 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

3.4. Identifying class dependencies

In the following Subsections, we present how the class dependencies were calculated with examples. We also present assumptions and decisions made during this task.

3.4.1. Logical Coupling

For each project we extracted the number of revisions, based on the tables built by CVSAnalY [4]. This task was a pure SQL extraction task, so it does not pose a time issue. For all revisions, we extracted the list of class pairs that were co-evolving in that revision and stored this data in a .CSV file. An example of the co-evolution data is provided in Table 3, detailing an excerpt of the Java classes that co-evolve in the *UrSQL* project in its 4^{th} revision. The first column shows the project name, the third and fourth columns show classes that were co-changed, through association rules.

Using the *arules* [5] library in the **R** [6] environment for association rule mining, we were able to compute the Confidence metric for each pair of classes with an established logical dependency (confidence > 0).

[4] https://sites.google.com/site/arnamoyswebsite/Welcome/updates-news/howtoinstallandruncvsanaly2inubuntu1110

[5] https://cran.r-project.org/web/packages/arules/index.html

[6] https://www.r-project.org/

Table 3: Co-evolution data for Project *UrSQL* (excerpt)

Project Name	Rev	class A	class B
UrSQL	4	UDO	Filio
UrSQL	4	UDO	Main
UrSQL	4	UDO	UrSQLController
UrSQL	4	UDO	UrSQLEntity
UrSQL	4	UDO	UrSQLEntry
UrSQL	4	Filio	UDO
UrSQL	4	Filio	Main
UrSQL	4	Filio	UrSQLController

3.4.2. Structural Coupling

While logical coupling is based on a time interval, structural coupling is defined for a specific time instant [2, 4]. Every snapshot of each project was parsed to extract the number of references 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 via the UNDERSTAND tool [7]. The references or calls from one class to another are and are used as a proxy for the structural relationship or coupling types, e.g., inheritance relationships [3].

Among the goals of this study is to understand the impact of logical coupling on structural coupling and vice versa. Therefore, we computed and used the number of references between any pair of coupled classes per project in the latest source code snapshot [4] in order to achieve this goal and to mitigate the threat of a change in the number of references between classes from one revision to another. Given that two coupled classes may have been co-changed multiple times in the past. Similar methodology is adopted by Geipel and Schweitzer who stated that structural dependencies between two classes somewhat stable from the creation of the younger class until the removal of one of the classes from an OO software. This process was automated using a Shell script we developed. As an example, Table 4 shows the number of operational calls between two Java classes (UrSQLController.java and UrSQLEntity.java) in the UrSQL project. The number of operational calls changes after two revisions as shown in column 5 and is sta-

[7] https://scitools.com/

ble during the last revisions shown in column 2; as at the time of extracting the data for this study.

Table 4: Coupling data for Project *UrSQL* (excerpt)

Name	Rev	Caller	Called	References
UrSQL	1	UrSQLController	UrSQLEntity	3
UrSQL	2	UrSQLController	UrSQLEntity	3
UrSQL	3	UrSQLController	UrSQLEntity	15
UrSQL	4	UrSQLController	UrSQLEntity	15
UrSQL	5	UrSQLController	UrSQLEntity	15
UrSQL	6	UrSQLController	UrSQLEntity	15
UrSQL	7	UrSQLController	UrSQLEntity	15

3.5. Evaluating the intersection of sets (RQ1)

Once pair-wise structural and logical dependencies were identified and the associated coupling values were calculated, we then built a spreadsheet per project based on the data with the following columns; LHS (antecedent), RHS (consequent), references, and confidence.

With this, we could start investigating our research questions. Firstly, we identify the distinct structurally dependent class pairs from the structural coupling data, as well as the distinct change dependent pairs from the co-evolution data.

Using a Shell script we developed, we could parse the data and identify the proportion of structural dependencies that involved non-logical dependencies; the proportion of logical dependencies that involved non-structural dependencies, and; the intersection set of pairs of classes that are both structurally and logically related.

3.6. Statistical tests – Spearman's Correlation (RQ2)

This Section describes the computation of statistical tests for RQ2. The intersection of coupling sets (from 3.5) is used to evaluate the relationship between the coupling types. All the values of the logical coupling strength (i.e., the *confidence* metrics); and all the values of the structural coupling strength (i.e., the number of *references* between classes), are pulled together, per pair of classes, per project. Given a project, we created two vectors, one with the values of 'number of references' between classes; the other with all

the values of co-change confidence between classes. The null hypothesis H_0 to be tested is as follows:

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

The correlation between the two vectors is evaluated using the Spearman's rank correlation coefficient [3]. Spearman's rank correlation, a non-parametric test was chosen because it is unlikely that either the structural or logical coupling values will have a normal distribution in each project.

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 [3, 30] (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 [31] 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.

For all the projects studied, we reject the null hypothesis at the 95% confidence level. In other words, if the rank correlation coefficient proves to be statistically significant at the $\alpha <= 0.05$ 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.* The results derived for all projects are exposed in Section 4.

One of the threats to the statistical validity of Yu's study [3] is the selection of the significance level. In that study, the chosen $\alpha = 0.1$ which might have resulted in a type I error – mistakenly rejecting a null hypothesis. To reduce this threat the plan for future work included increasing the confidence level to 95% (reducing the α value to 0.05) for more accuracy which we have done in this study to mitigate this threat.

*3.7. Identifying **stability** level of structural links (RQ3)*

In order to answer research question Q3 from Table 1 above, we need to identify (i) which pairs of classes are structurally coupled, (ii) how many times they co-changed (as depicted in Figure 3) and (iii) assign a level of *stability* to each pair.

As an example, for two classes A and B that are structurally and logically coupled we identified their level of structural stability by means of the following steps:

- we counted the number of revisions when the pair shows a structural link, and we named it the *life cycle* of that link, and for that pair;
- we divided this life cycle in two, so to obtain two halves of the life cycle;
- we identified in which half the pair also showed a co-change. Only the co-changes relating to the structural links between the classes are considered, i.e., this co-change affects the variable `gv` or the function `func()` or both as in Figure 3;
- we made a decision on the stability of a class pair based on the resulting co-change pattern.

This procedure is visually summarised in Figure 4. The class pair $A \rightarrow B$ is structurally coupled (shown by the x symbols) for 6 revisions (its life cycle) and co-changed (shown by the o symbols) in four revisions, three in the first half and once in the second half. It is noteworthy that this life cycle or total number of revisions may vary depending on the classes involved. In addition, this is an exploratory study, thus we carried out a median split [32] on the life cycle into two equal halves where possible to mitigate bias in either direction of the split in the absence of a parametric model. The median split procedure has been frequently used in prior research [33] to convert continuous variables into categorical variables for further analyses and interpretation.

This is to enable us investigate homogeneous portions of the life cycle of dependencies in terms of number of revisions across the splits [34] using the standard held-out (HO) splitting method (splitting data into two halves for empirical evaluation) [35]. Using a different split for example one quarter on one hand and three quarter on the other hand provides a higher chance that the results will be skewed the results in one direction. The chance of having minimal levels of maintenance activity within the one quarter split will be higher and will result in a large number of either unstable pairs or static stable pairs.

3.7.1. Drawing Scenarios of Stability

Using Figure 4 as a visual example, we identified the following 5 scenarios (also pictorially drawn in Figure 5):

1. *Static stable pairs*: this set is composed of all the pairs of classes that indeed have a coupling link between them, but they do not co-evolve.

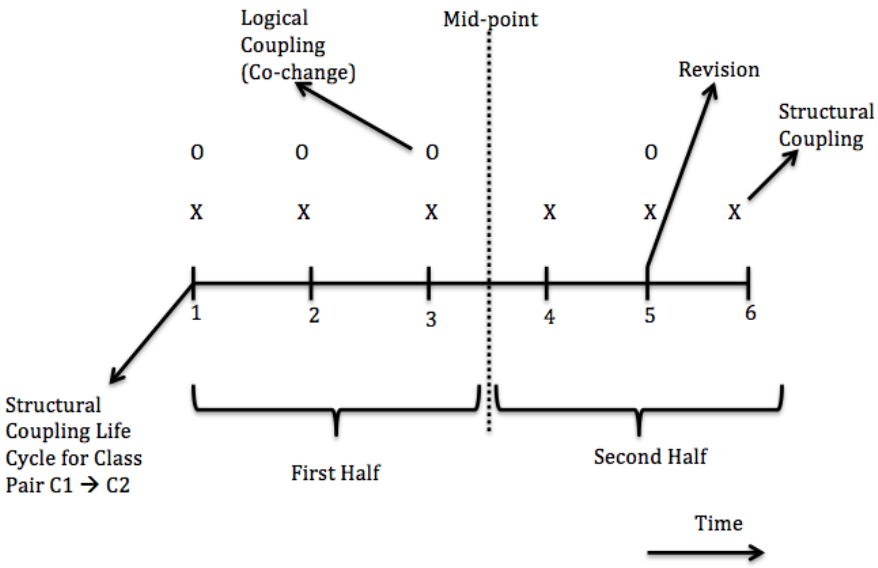

Figure 4: Evaluating the structural stability for a pair of classes, using structural coupling (x) and co-change (o)

This is a common scenario for software projects, so our contribution is to clarify its relevance in a large pool of projects.

2. *Stable pairs*: the pairs in this scenario only co-change in the first half of the life cycle. This points to the classes that need maintenance in a limited number of revisions, after establishing a link between them.
3. *Partially stable pairs*: the pairs in this scenario are co-changed in both halves, but with a majority of the co-changes in the first half. This is an interesting scenario and is worth more investigation, whereby the pair of classes require frequent maintenance. While this scenario points to more maintenance needed into these classes, overall we still cluster them in the somewhat *structurally stable* class pairs. The majority of this further maintenance is needed for a limited period.
4. *Partially unstable pairs*: the pairs in this subset are either equally co-changed in both halves, or with a majority of the co-changes in the second half. This points to more maintenance needed, and not only early on: even after having established the structural link, effort is still needed later on in the life cycle.
5. *Unstable pairs*: the class pairs in this scenario are co-changed only

in the second half of their life cycle. Changes to the structural link between these classes materialise not when a coupling link is established between them but only in a future moment.

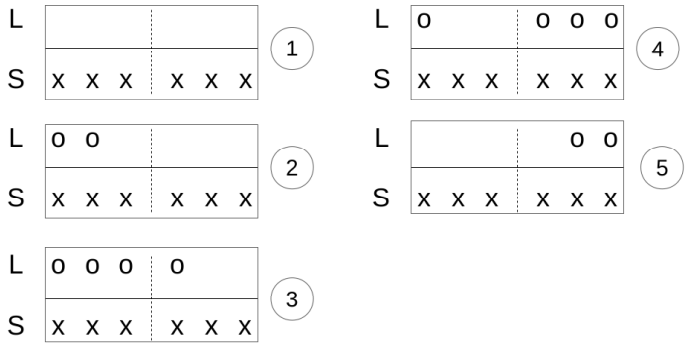

Figure 5: Examples of the 5 stability scenarios. (KEY: L = Logical Coupling; S = Structural Coupling)

We argue that scenarios 1, 2 and 3 jointly represent the subset of stable (to a various degree) pairs in a system. Conversely, scenarios 4 and 5 can be linked to an overall instability of a pair of classes. The scenarios, and the evaluation of their stability, can be used to investigate the quality of software systems and their architecture. For example, a poorly designed system with tightly coupled classes will require frequent maintenance to a majority of the coupled classes given a new requirement [1].

4. Results

Following the methodology outlined above, this Section presents the results of the three analyses, as performed on the selected projects. The aim is to answer the three research questions outlined in Table 1.

4.1. RQ1. Is there a «directional» relationship between structural and logical coupling?

To answer this question, the aim was to get a view of the overlapping or intersection set of structural and logical class dependencies in OO software projects. Once the two sets of coupling are computed per project, the intersection set of class dependencies represents the proportion of pairs of classes

that are both logically and structurally related. Depending on the size of the two sets, the Venn diagram could be far from symmetric.

From Table B.7 in Appendix B we know, for example, that the project with ID=31 has a proportion of 68% of its structural class dependencies including logical dependencies (as shown in the 6^{th} column). On the other hand, only 21% of the pairs that co-changed include structural dependencies (number of operational calls or references > 0), in the same project. This is a recurring pattern: in a majority (81%) of the projects, we have evidence to indicate that very often, structurally related classes involve logically related classes (64 of 79 projects).

In 5 of these projects, **all** the structural dependencies are reflected into logical dependencies. In both Venn diagrams (left and right) in Figure 6, the smaller circle represents the set of structural dependencies while the larger circle represents the set of logical dependencies.

Using the Venn diagram on the left (weighted) in Figure 6, all the coupled pairs of classes in the *jbandwidthlog* project (project ID = 97) need also co-changes. On the flip side, not all the pairs that co-change are structurally coupled.

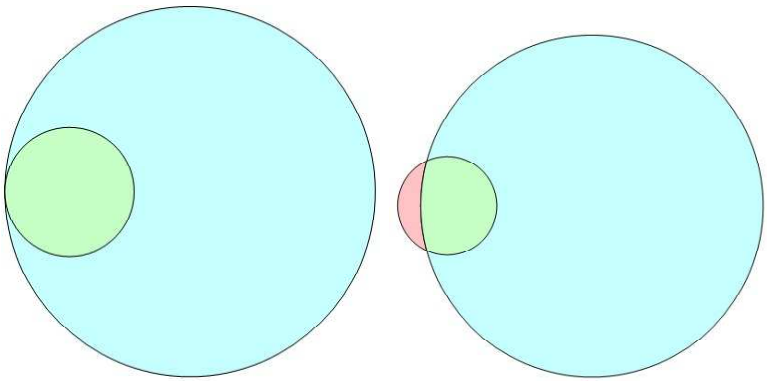

Figure 6: Venn Diagrams (weighted) showing the two sets of coupling in two scenarios: project ID=97 (left) and project ID=69 (right)

The second most common scenario identified in the results is illustrated using the Venn diagram in Figure 6 (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 conducive of structural coupling.

On the other hand, a majority of the projects show evidence to indicate that very often, logically related classes do not involve structurally coupled classes. In a study on the relationship between logical and structural coupling, Oliva and Gerosa classified the logical coupling measurements as follows: 0 - 0.33 (low), 0.33 - 0.66 (medium), 0.66 - 1.0 (high) [2].

Applying a similar categorization on the percentages in Table B.7 (in Appendix B), allows us to infer that the proportion of logically related classes that involve structurally related classes OO software is relatively low [8] in all projects studied, ranging from 1% to 44%. With over half (63) of the 79 studied projects with 20% or below.

This confirms the directional relationship (structural coupling → co-evolution of classes) between structural and logical coupling, as identified by Yu [3] and shown in Figure 1. Therefore, to answer RQ1, we conclude that there is a directional relationship between structural and logical coupling in OO software, and for the majority of projects in our studied sample.

Figure 7 shows two summary box-plots with the following percentages:

$$CSD(\%) = \frac{Structural \cap Logical}{Structural} \tag{1}$$

$$CLD(\%) = \frac{Structural \cap Logical}{Logical} \tag{2}$$

The Co-changed Structural Dependencies ratio (CSD) is the percentage of structurally coupled pairs that have also been co-changed with respect to all the structurally coupled pairs per project. *Structural* implies the set of class pairs with source code dependencies between them, while *Logical* implies the set of classes observed to have co-changed in the past.

The Coupled Logical Dependencies ratio (CLD) is a similar percentage, but evaluated with the co-change sets. The two box plots are exemplary of a common pattern: the median CSD ratio is around 80%, meaning that, for all the projects, most of the structurally coupled pairs also co-evolve. On the other side, the median CLD shows that, for most projects, very few co-changing pairs are also coupled.

Around 80% of structural class dependencies include logical dependencies but not vice versa

[8] compared to the proportion of structural dependencies belonging to the intersection set

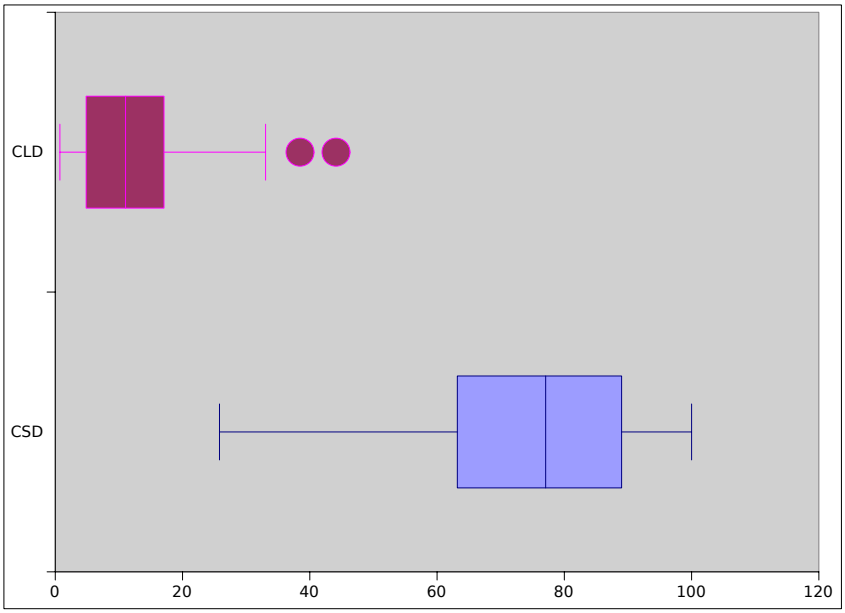

Figure 7: CLD and CSD Percentages per OSS Project (KEY: CSD = Co-changed Structural Dependencies; CLD = Coupled Logical Dependencies)

4.2. RQ2. Is there a «linear» relationship between structural and logical coupling?

To answer this research question, we used the method outlined in Section 3.4 and the statistical approach shown in Section 3.6. Using the Spearman's rank correlation, we tested for the null hypothesis H_0: *No linear relationship between the strengths of logical and structural class dependencies in OO software* (at $\alpha = 0.05$).

Figure 8 shows the generic correlation outcomes, alongside the p-values derived from the Spearman's rank correlation analysis computation. As visible, there is a significant correlation (i.e., p-value $<= 0.05$) in less than 10 projects. In a handful of projects we observed an insignificant (i.e., p-value > 0.05) negative Spearman's correlation between the structural coupling strength (i.e., references between classes) and their co-evolution strength (i.e., confidence), while the majority of projects show an insignificant positive correlation coefficient.

Differently from [3], where a correlation (albeit at $\alpha <= 0.1$) was indeed found between references (i.e., structural coupling) and confidence (i.e., co-change), we do not find a strong evidence to fail to reject H_0. The results of

the correlation tests are visible in Figure 8. The outcome of the Spearman's rank correlation was concluded considering the overall pool of projects, rather than the single projects alone. This is because, we cannot generalize the findings derived from a small subset or a minority of projects from our sample.

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

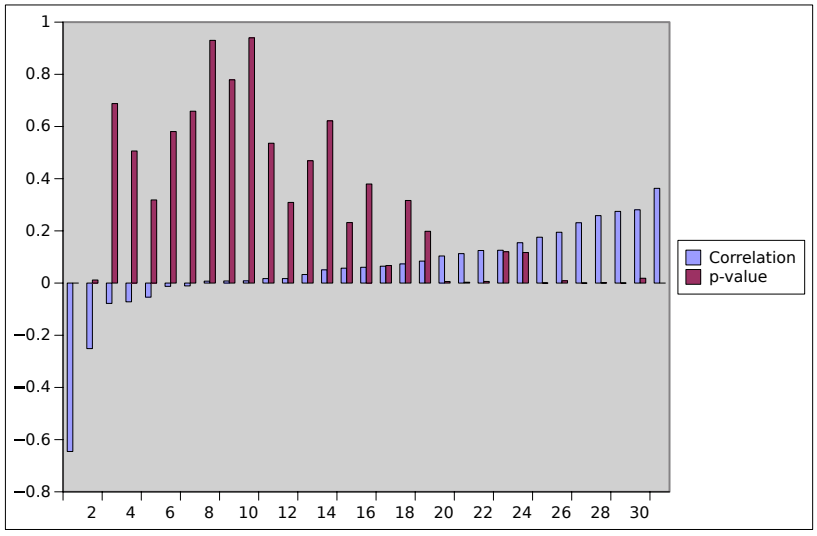

Figure 8: RQ2 - Spearman's Rank Correlation (correlation results with p-values)

4.3. RQ3. What is the proportion of stable pairs of classes in a software system?

To answer RQ3, we evaluate the overall structural stability of the studied sample of projects by adopting the five stability scenarios from Section 3.7. In the same section, we described the steps taken to achieve the structural stability using an example pair of structurally and logically classes A and B.

Firstly, in this section, using the following projects; *2dTetris* (ID=1), *monome-pages* (ID=142) and *fyllgen* (ID=60) as examples, we briefly discuss class pairs that have been identified as belonging to each of the five stability scenarios from these projects. In Table 5, we present a structural class dependency per structural stability scenario alongside the number of times the class pair have undergone co-changes in the first and second halves of their structural coupling life cycle and discuss these further afterwards.

Table 5: Structural stability scenarios - worked example

Caller	Called	Stability	# Coupling	1st half	2nd half
MidiPlayer	Statistics	Static stable	222	0	0
NetGame	GameImpl	Stable	258	5	0
TetrisMainWindow	FieldGraphic	Partially stable	120	3	2
MIDITriggersPage	MonomeConfiguration	Partially unstable	249	13	14
GUI	Family	Unstable	49	0	7

1. *Static stable pairs*: the class pair MidiPlayer.java and Statistics.java are structurally linked but they never co-change.
2. *Stable pairs*: NetGame.java and GameImpl.java only co-change in the first half of their life cycle. These classes have only been co-changed in 5 revisions of the *2dTetris* project, and all in the first half of their coupling life cycle.
3. *Partially stable pairs*: the classes TetrisMainWindow.java and FieldGraphic.java in the *2dTetris* project belong to this scenario. The pair were are co-changed in both halves of their structural coupling life cycle, but with a majority of the co-changes in the first half.
4. *Partially unstable pairs*: the classes MIDITriggersPage.java and MonomeConfiguration.java in the *monome-pages* project belong to this category. The class pair equally co-changed in both halves of their structural life cycle but with a majority of the co-changes occurring in the second half.
5. *Unstable pairs*: GUI.java and Family.java in the *fyllgen* project are examples of structurally linked classes in this scenario. The classes are co-changed only in the second half of their life cycle. Modifications to these classes materialise not when an operational link is established between them but only in a future moment.

Repeating the same analysis for all the projects, Appendix A shows the percentages of pairs falling in each scenarios, per project (Table A.6). Figure 9 further summarises the overall sample of 79 studied OO software projects. Box-plots are used for the 5 scenarios.

Looking at the box-plots in Figure 9:

- only around 10% of class pairs are unstable pairs. However, one project in particular (*fyllgen*) contains 57% of unstable pairs of classes, although it is an outlier.
- As a result of the many outliers, the median value in the "unstable" scenario is significantly lower than the average.
- The "partially stable" and "partially unstable" clusters are more compact than other clusters, and show less variability (around 8% and 10%, respectively, on average).
- The percentages of stable and static-stable pairs show the higher variability, given the studied sample of projects.
- All the projects show more stable pairs than unstable apart from the outlier project (*fyllgen*).

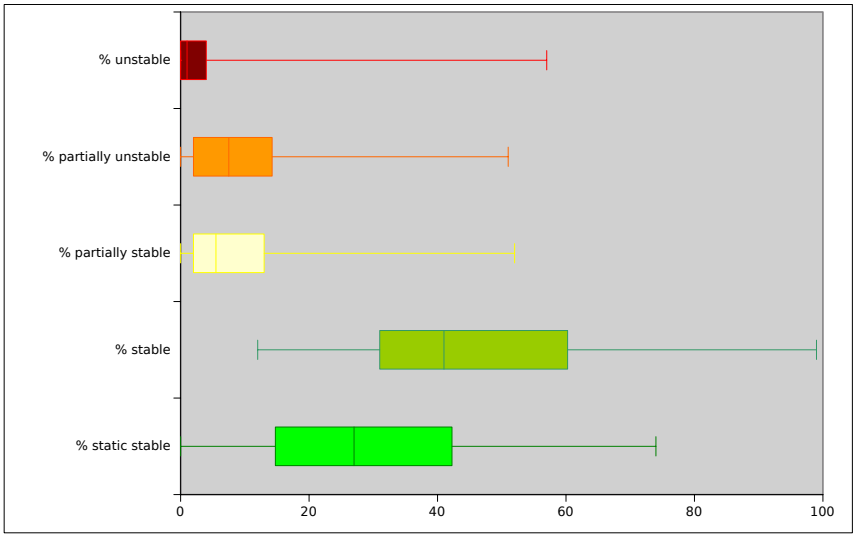

Figure 9: Summary of Structural Stability of Studied Sample of Projects

The studies systems in the sample show a larger set of stable pairs of classes than unstable

5. Discussion

In this study, we have conducted a large scale empirical study on the (i) linear relationship between the structural and logical dependencies of pairs of Java classes (ii) existence or non-existence of a directional relationship between structural and logical coupling [3] and (iii) overall evaluation of the structural stability of OO software projects. Below we discuss the findings reported, and put them in perspective for the software maintenance field.

5.1. Directional relationship between the structural and logical coupling of OO software classes

Our results from analysing a different sample of OSS projects from a different repository to the one studied by Geipel and Schweitzer [4] have showed 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, they have identified that the proportion of change dependencies is always larger than the proportion of structural dependencies [5, 2].

The intersection of these two sets is particularly important: in 64 out of the sample of 79 studied projects, between 60% and 100% of the structurally coupled pairs co-change once or more. Differently from other research results, that tend to highlight the 80-20 Pareto distribution in most of the metrics on single software artifacts (complexity [36], defect density [37], number of changes [38]), we have detected that pairs of structurally coupled classes do not follow such distributions.

5.2. Linear relationship between the structural and logical coupling of OO software classes

The results of the analysis of RQ2 are presented in Section 4. When using $\alpha <= 0.1$, Yu [3] found a correlation between the structural and the logical coupling. On the contrary, using $\alpha <= 0.05$, we have shown that there is no correlation between the two: a stronger coupling between two classes is not a predictor of the likelihood of more changes to that pair of classes. While computing the correlation coefficients per project, the outputs also showed that using $\alpha <= 0.1$ will not have any effects on the results.

A reason for this could be the fact that software architectures change, a certain class A may stop using features from another class B after a while or the class B might be removed [4]. Ripple effects can also play a major role:

apart from the $A - B$ link, one should consider all the links around the A and B classes alone, as visible in Figure 10.

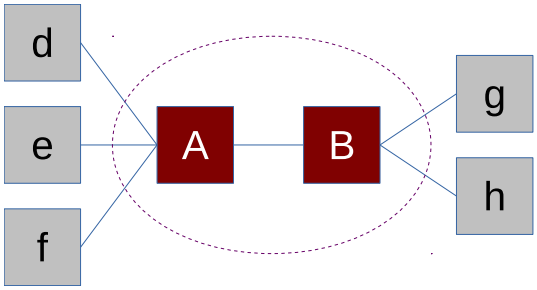

Figure 10: Effect of networks

Changes to the d, e or f classes, connected to A alone, can have ripple effects on the $A-B$ coupling link. Similarly, changes to g and h can influence their link to B, and in turn the $A-B$ link too. This effect was not investigated in this paper, but it is likely to play a role in how the maintenance efforts are linked to co-changes. Previous research [5] has shown that ripple effects are propagated across the path of structural dependencies in OO software.

5.3. Object-Oriented Software Structural Stability

To achieve the final goal of this study, we finally investigated the impact of time on the structural link between class pairs. We posit that after the addition of a structural link between two classes, the need of co-change should degrade over time. In doing so, we defined 5 levels of stability to define a coupling link, from "statically" stable to unstable.

From the results gathered, a vast majority of the coupling links are stable over time: once inserted they do not need major maintenance work. This is in line with practitioners' advice: software systems should be built with low coupling and high cohesion to improve comprehension, reuse and maintenance [1].

The stability to changes of a software system draws a similar scenario to structural engineering: too many interactions between components (i.e., coupling) affect maintenance (i.e., renovation) of a building [18] and should be kept to a minimum. The *types* of renovations in the structural engineering discipline are also similar to OO software co-changes (i.e., classes). Slaughter [18] outlined the types of changes that can be expected over the long term in buildings and these include: (i) change in functions (i.e., change in class

functionality [39, 40, 41]), and (ii) flow, or the movement within buildings (i.e., change in the access scope and mode for any datum in software systems [41, 42]). Finally, the *nature* of component interactions influence the flexibility of building structures to the different maintenance types [18, 43, 44].

On the other hand, a clear definition of the instability (to changes) of specific pairs of classes has evident benefits. The skewness of changes to single classes is evident from past studies; in our work we posit that links between classes should be considered too, since a small set of them requires more maintenance than other parts.

5.4. Impact of Findings

Based on the findings of this study, we can infer that the co-evolution of software classes are partly brought about by source code dependencies, thus a directional relationship exists between the system architecture and the co-evolution of software classes. It can also be inferred that since not all the logical dependencies include structural dependencies, logical dependencies could be related to other forms of software dependencies, for example semantic coupling [25].

5.4.1. Semantic Coupling

According to Bavota *et al.* [45] "*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, and cannot be easily derived by only looking at structural relationships, such as method calls*".

Other researchers in the software evolution and dependency domain have identified that semantic coupling metrics can outperform structural metrics in identifying classes that might be impacted by a given change request [46] and have combined semantic and logical coupling metrics in change impact analysis [47, 48]. However, there is still the need to study the interplay between semantic and logical coupling in OO software [2, 5, 49].

5.4.2. Co-Testing

In Section 1 we identified several application in software engineering based on the interplay between structural coupling and co-evolution of OO software classes. These include prediction of software changes; inferring co-evolution from structural dependencies and; focusing testing effort. Our results have

shown that structural coupling in most cases will lead to co-evolution of classes, therefore related classes need to be co-tested after modifications.

5.4.3. Dependency Management Tool-chain

According to Oliva and Gerosa "*if the overlapping between structural and logical coupling is large, then structural and logical dependencies can be used interchangeably as input to dependency management methods and tools. On the other hand, if the overlapping is actually small, then it would be necessary to conceive and develop novel dependency management methods and tools that incorporate both kinds of dependencies*" [2]. Our results have supported the fact that the overlapping between both types of dependencies is not large, thus software management tools will need to draw insights from both types of dependencies.

5.4.4. Co-change Prediction

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 [4] because the proportion of change dependencies is always larger than the proportion of structural dependencies [5, 2]. On the other hand, Oliva and Gerosa state that using the structural coupling information between pairs of classes to predict unplanned future co-changes is a more realistic objective [5]. Our contribution adds to these past works: the prediction objective is realistic, but only with the support of other coupling metrics, e.g., semantic coupling. This is because the overlapping between structural and logical dependencies is not large.

Finally, Geipel and Schweitzer have stated that the question about the causes of change propagation has been overlooked by many researches in favor of a predictive approach [4]. Our results and contributions on a large sample of 79 OSS projects provide statistical backing for the results in a study carried out by Abdeen *et al.* [50]. They performed inter-system and intra-system change impact prediction using structural, semantic dependencies and a combination of both and compared results. They identified that using semantic coupling produces better recall values, in particular, in the intra-system scenario. They state that an 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 outperforms semantic dependencies. Our results have statistically shown the

absence of a linear correlation between the degree of the structural and logical coupling of classes in OO software. In addition, Only a minority of the co-changed classes can be accounted for by structural dependencies. Therefore, as Abdeen *et al.* have shown; using structural dependency information alone will not yield a high precision or recall in co-change prediction compared to semantic coupling.

6. Related Work

Structural and logical (evolutionary) dependencies are at the core of software engineering. In the following Section we are summarising the main results of related studies on both aspects separately, and when studied jointly.

6.1. Structural Coupling

Structural coupling (simply called "coupling" in some studies [3, 51, 52, 53]) is still considered to be an imprecise measure of software complexity [51]. Many researchers have empirically investigated and identified the relationship between coupling and the external quality factors of software products such as fault-proneness and maintenance [54, 45], change impact analysis [13, 46, 55, 56], re-engineering, reuse, change propagation, and clone management [45].

These studies proposed various structural dependency metrics which add to the large number of metrics that already exist. Various attempts have been made to address this problem by developing frameworks for coupling measures to generate a consensus in the software engineering community (i.e., defining proper measures for specific problems) [57, 58, 59].

6.2. Logical Coupling

In comparison to the broad research on structural coupling, the study of logical coupling, evolutionary or change dependencies [3, 2, 6] has just begun a few years ago because of the advances in data mining techniques [3] used to extract co-evolution data. However, despite its short history, there have been several interesting studies published with promising results.

Ying *et al.* [20] proposed an approach to predict source code changes by mining change history of software systems. Zimmermann *et al.*[6] applied data mining to version histories in order to guide programmers along related changes using the idea that "Programmers who changed these functions also changed...." [6]. Given a set of existing changes, the mined association rules 1) suggest and predict likely further changes, 2) show up item coupling that

is undetectable by program analysis, and 3) can prevent errors due to incomplete changes.

6.3. The Link Between Structural and Logical Coupling

For most of the studies described in Subsection 6.1 and 6.2, the study of either structural coupling or co-evolution of classes was done separately, at the source code level (coupling), or based on CVS (Concurrent Versions System) release history data (co-evolution) which reveals the evolutionary dependencies between software entities [10]. Differently from previous studies, we have empirically explored the direct influence of structural coupling on logical coupling and vice versa in different ways: the correlation between the structural and logical coupling strengths between classes and the overlapping of the coupling types at the class level of granularity. There have been several studies that have been performed to understand the relationship between the co-evolution of classes and their structural coupling [60, 61, 3, 15, 9, 2, 29, 4, 5], and we discuss them according to their year of publication in ascending order.

Gall *et al.* [9] were the first to use co-evolution to represent structural coupling. They developed a technique called CAESAR for detecting change patterns and applied it to a large Telecommunication Switching System with a 20-release history. Their approach identifies evolutionary 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 [9]).

Zimmermann *et al.* [15] analysed the revision history of individual classes and functions to detect the fine-grained coupling (they noticed that classes with strong co-evolution also have strong structural coupling but did not provide empirical evidence). In this study, we adapt the metrics (i.e., support and confidence) as proposed by Zimmermann *et al.* [15] to measure the strength of association rules in our sample. We have also shown that structural and logical coupling strengths have minimal influence on each other.

Fluri *et al.* [29] investigated the degree to which co-changes are caused by structural changes (source code/structural coupling) 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. In this

study we have shown that structural dependencies will usually include logical dependencies but not the other way round.

Yu [3] conducted a study on 12 Linux kernel modules, comparing 12 pairs of co-evolution data and 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 classes induced via the system architecture have noticeable effects on class co-evolution. Although Yu studies only 12 Linux classes, the study is the most similar to ours and we have studied a large sample of 79 OSS projects. As mentioned in Section 3.6, in Yu's study [3] one of the threats to the statistical validity is the selection of the significance level. The chosen $\alpha = 0.1$ might have resulted in a type I error – mistakenly rejecting a null hypothesis. To reduce this threat they planned to increase the confidence level to 95% (reducing the α value to 0.05) for more accuracy which we have done in this study and achieved different results. We also identified that using $\alpha = 0.1$ will not change our results or conclusions.

Oliva and Gerosa [2] analyse Java files of the first 150 thousand commits from apache software repository (ASF) 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 and 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. However, differently from our study: the number of structurally coupled pairs of classes used in their study was computed based on an estimate. They derived the number of coupled pairs of classes by multiplying the average CBO (number of classes each class is structurally coupled to) by the distinct number of classes and for this reason, they acknowledge that their results are not really reliable. Oliva and Gerosa [2] also suggests extending their study to other OSS repositories and in this study, the subject systems were taken from the GoogleCode repository. Thus it is important that this study is conducted using a different sample of projects as well as methodology.

Recent studies [2, 5, 4] have shown that it is possible that both structural and logical coupling are caused by other types of software dependencies (e.g., conceptual dependencies). An example of conceptual dependencies between class methods is presented in [13], where the conceptual coupling values for

the pair `addShape()` and `removeShape()` is 0.78. The conceptual coupling value is between 0 and 1 and it is a symmetric metric, i.e., the values of (`addShape()`, `removeShape()`) and (`removeShape()`, `addShape()`) are the same. Both methods contain similar terms such as `canvas`, `frameset`, and `shape`, that contribute to the conceptual similarity between these methods.

In a study including 16 OSS projects, using Pearson correlation, Beck and Diehl [60] conducted pairwise correlations on various software coupling concepts to identify whether pairs of classes coupled by one concept are also coupled a second concept. Interestingly, they found no correlation between structural and logical coupling as well as between semantic and logical coupling. However, they found a correlation between semantic coupling and code ownership for obvious reasons; latent semantic indexing (LSI) is an information retrieval technique adopted to identify the degree to which the underlying meanings of words or terms in different documents (e.g., identifiers and comments in classes) are related and as such the semantic coupling of class pairs measured using LSI [25, 47] is based on the presence of similar terms present in source code and code ownership is based on the concept that two classes are related if they share the same author; and author names are embedded in the source code of both classes. They also identified a correlation between ownership coupling and logical coupling. An explanation is that both are based on the check-in information. In this study, we report the results derived from a large sample of OSS projects using a different methodology to identify the interplay between structural and logical coupling of Java classes.

Geipel and Schweitzer [4] analyze the link between structural dependency and the co-change frequency of OO software classes. Their study takes into consideration 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 by [5]. In addition, 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 [2] and other studies [4, 21, 62], Oliva and Gerosa [5] conduct 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. Their results indicated that in general, it is more likely that two software artifacts will not co-change just because one depends on the other. However, the rate with which an artifact co-changes with another is higher when the former structurally depends on the latter. This rate becomes higher if the dependencies are tracked down to the low -level entities that are changed in commits. This implies, for instance, that developers should be aware of dependencies on methods that are added or changed, as these dependencies tend to propagate changes more often.

7. Threats To Validity

In this Section we present the threats to validity of this study, dividing them in *external*, *internal* and *construct* threats.

External validity. This paper presents the results of an empirical analysis that should be applicable to all OSS projects. We cannot generalize our findings on any other sample of OSS projects, or from any other repository. Nonetheless, in order to make the findings from our study more generalisable and representative of OSS projects, we have carried out our analysis on a large random sample of projects, with different sizes as well as different number of past changes.

Internal validity. We acknowledge the fact that support and confidence values of association rules could produce misleading results [2]. 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).

In addition, our method for partitioning the structural coupling life cycle of coupled pairs of classes when answering RQ3 is not efficient in some cases. That is cases where a pair of classes are coupled in an odd number of revisions, we use rounded up values to determine the number of revisions in the first and second half. For example, only in three revisions. The mid-point should be just after revision 1.5, thus there will be two ($1.5 \rightarrow 2$) revisions on one half and only one in the second half.

Another threat to the validity of the results for RQ3 is the migration of projects between open source forges. We acknowledge that some of the

projects might have been migrated from one repository to another. This could mean that parts of the structural life cycle of some class pairs are not migrated. However, to mitigate this threat we examined the initial commit logs of 10 projects in our studied sample (project ID = 8; 26; 28; 51; 56; 69; 97; 109; 152; and 172 in Table B.7) by means of the CVSAnaly tool and parsed these to identify whether the developers' commit messages indicate any migrations from another repository.

Table C.8 in Appendix C shows the commit message found for the initial commit in 10 OSS projects (randomly selected from out pool of 79). Out of these projects, only one project (e.g., *bluecove*) shows a migration from the SourceForge repository, and reflected in the history log.

Construct validity. The scope of our sample of projects was limited to open-source software projects written in the Java programming language (object-oriented), thus we encourage investigating projects written in other programming languages and non-object-oriented software projects.

The Fisher Exact Test tests for the dependence between two categorical variables. However we have not relied on that test in this study to identify whether there is a directional relationship between the structural and logical coupling of OO software classes because while it tests for a dependence or association it does not indicate the direction.

To assess the presence of a linear correlation between the strengths of the structural and logical coupling between classes we adopted the Spearman's rank correlation coefficient. This is because it does not assume a normal distribution and we cannot guarantee that the strength of the coupling between classes will follow a normal distribution across the history of software projects. Notwithstanding the test has its disadvantages: it takes into consideration the ranked order of the structural coupling metrics and not the values themselves. In other words, as long as the order of the structural coupling metrics remain the same the coefficient will stay the same.

To determine whether the correlation coefficients will remain the same across different versions of the studied systems, for a subset of systems we have computed the correlation using structural coupling metrics from earlier versions prior to the last version and all the correlations remained the same. Therefore, we deduce that structural coupling changes from version to version for some individual class pairs may not affect the rank correlation of Spearman's correlation test [3].

8. Conclusion and Future Work

We have conducted a three-fold empirical study on a sample of 79 open-source software projects to identify if there is a relationship between structural dependency and co-change of object-oriented (OO) software classes. The number of projects used for this study is larger than those used by previous studies. More importantly, and differently from previous studies, our sample considers every single revision that each project underwent, instead of only one snapshot.

Firstly, we investigated the interplay between structural and logical coupling to identify whether there is a directional relationship besides a linear relationship between the two. Results pointed to the presence of a small overlapping between structural and logical dependencies in a majority of the software projects. However, we noticed a higher likelihood of structural dependencies leading to co-evolution of classes but a small chance of being able to infer structural coupling from co-evolution.

Secondly, using Spearman's correlation we investigated whether a linear relationship exists between the structural and logical coupling strengths of pairs of OO software classes. Results from this investigation revealed that there is in fact no strong evidence to suggest that a linear relationship exists between the strengths of different types of couplings. A stronger structural coupling does not imply a higher co-change likelihood.

Thirdly, we noticed a significant rate of *structural stability* of coupled class pairs in the overall sample of projects studied. Coupling links are inserted between classes and need a limited maintenance. The measurements used clearly highlight the presence of a set of unstable links, that cause repeated co-changes.

We have discussed the impact of our findings on software maintenance in Section 5.4 and as future work, we plan to carry out studies on the same sample of projects, to detect whether there are linear and directional relationships between *semantic* and logical dependencies. The rationale being that if such a relationship exists, semantic coupling metrics can be used to directly inform practitioner about potential co-changes of classes in OO software projects. In addition, semantic coupling metrics will be used to inform or predict the strength of the logical dependencies between classes without the need to analyze historical data of software projects thus reducing the computation time and efforts required in the detection of logical dependencies via mining software repositories (MSR).

9. References

[1] B. Henderson-Sellers, L. L. Constantine, I. M. Graham, Coupling and cohesion (towards a valid metrics suite for object-oriented analysis and design), Object Oriented Systems 3 (3) (1996) 143–158.

[2] G. A. Oliva, M. A. Gerosa, On the interplay between structural and logical dependencies in open-source software, in: Software Engineering (SBES), 2011 25th Brazilian Symposium on, IEEE, 2011, pp. 144–153.

[3] L. Yu, Understanding component co-evolution with a study on linux, Empirical Software Engineering 12 (2) (2007) 123–141.

[4] M. M. Geipel, F. Schweitzer, The link between dependency and cochange: empirical evidence, Software Engineering, IEEE Transactions on 38 (6) (2012) 1432–1444.

[5] G. A. Oliva, 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.

[6] T. Zimmermann, A. Zeller, P. Weissgerber, S. Diehl, Mining version histories to guide software changes, Software Engineering, IEEE Transactions on 31 (6) (2005) 429–445.

[7] M. D'Ambros, M. Lanza, M. Lungu, Visualizing co-change information with the evolution radar, Software Engineering, IEEE Transactions on 35 (5) (2009) 720–735.

[8] I. S. Wiese, R. T. Kuroda, R. Re, G. A. Oliva, 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, Springer, 2015, pp. 3–12.

[9] H. Gall, K. Hajek, M. Jazayeri, Detection of logical coupling based on product release history, in: Software Maintenance, 1998. Proceedings., International Conference on, IEEE, 1998, pp. 190–198.

[10] H. Gall, M. Jazayeri, J. Krajewski, Cvs release history data for detecting logical couplings, in: Software Evolution, 2003. Proceedings. Sixth International Workshop on Principles of, IEEE, 2003, pp. 13–23.

[11] M. D'Ambros, M. Lanza, R. Robbes, On the relationship between change coupling and software defects, in: Reverse Engineering, 2009. WCRE'09. 16th Working Conference on, IEEE, Lille, France, 2009, pp. 135–144.

[12] I. Wiese, R. Kuroda, R. Ré, R. Bulhões, G. Oliva, M. Gerosa, Do historical metrics and developers communication aid to predict change couplings?, Latin America Transactions, IEEE (Revista IEEE America Latina) 13 (6) (2015) 1979–1988.

[13] H. Kagdi, M. Gethers, D. Poshyvanyk, Integrating conceptual and logical couplings for change impact analysis in software, Empirical Software Engineering 18 (5) (2013) 933–969.

[14] M. D'Ambros, M. Lanza, M. Lungu, The evolution radar: Visualizing integrated logical coupling information, in: Proceedings of the 2006 international workshop on Mining software repositories, ACM, 2006, pp. 26–32.

[15] T. Zimmermann, S. Diehl, A. Zeller, How history justifies system architecture (or not), in: Software Evolution, 2003. Proceedings. Sixth International Workshop on Principles of, IEEE, 2003, pp. 73–83.

[16] L. Prechelt, An empirical comparison of c, c++, java, perl, python, rexx and tcl, IEEE Computer 33 (10) (2000) 23–29.

[17] F. Beck, 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, ACM, 2011, pp. 354–364.

[18] E. S. Slaughter, Design strategies to increase building flexibility, Building Research & Information 29 (3) (2001) 208–217.

[19] M. Holmes, Common Renovation Mistakes and How to Avoid Them - Homebuilding & Renovating kernel description (2008).
URL *https : //www.homebuilding.co.uk/common − renovation − mistakes − and − how − to − avoid − them/*

[20] A. T. Ying, G. C. Murphy, R. Ng, M. C. Chu-Carroll, Predicting source code changes by mining change history, Software Engineering, IEEE Transactions on 30 (9) (2004) 574–586.

[21] A. E. Hassan, R. C. Holt, Predicting change propagation in software systems, in: Software Maintenance, 2004. Proceedings. 20th IEEE International Conference on, IEEE, 2004, pp. 284–293.

[22] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, Predicting the probability of change in object-oriented systems, Software Engineering, IEEE Transactions on 31 (7) (2005) 601–614.

[23] R. Malhotra, A. J. Bansal, Cross project change prediction using open source projects, in: Advances in Computing, Communications and Informatics (ICACCI, 2014 International Conference on, IEEE, 2014, pp. 201–207.

[24] R. Vanciu, V. Rajlich, Hidden dependencies in software systems, in: Software Maintenance (ICSM), 2010 IEEE International Conference on, IEEE, 2010, pp. 1–10.

[25] D. Poshyvanyk, A. Marcus, The conceptual coupling metrics for object-oriented systems., in: ICSM, Vol. 6, 2006, pp. 469–478.

[26] D. Cruz, T. Wieland, A. Ziegler, Evaluation criteria for free/open source software products based on project analysis, Software Process: Improvement and Practice 11 (2) (2006) 107–122.

[27] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, D. Damian, An in-depth study of the promises and perils of mining github, Empirical Software Engineering 21 (5) (2016) 2035–2071.

[28] B. A. Romo, A. Capiluppi, T. Hall, Filling the gaps of development logs and bug issue data, in: Proceedings of The International Symposium on Open Collaboration, ACM, 2014, p. 8.

[29] B. Fluri, H. C. Gall, M. Pinzger, Fine-grained analysis of change couplings, in: Source Code Analysis and Manipulation, 2005. Fifth IEEE International Workshop on, IEEE, 2005, pp. 66–74.

[30] R. R. Pagano, Understanding statistics in the behavioral sciences, 6th Edition, Wadsworth-Thomson Learning, Australia;United Kingdom;, 2001.

[31] A. P. Field, Discovering statistics using SPSS: and sex and drugs and rock 'n' roll, 3rd Edition, SAGE, London;Los Angeles;, 2009.

[32] D. Iacobucci, S. S. Posavac, F. R. Kardes, M. Schneider, D. Popovich, Toward a more nuanced understanding of the statistical properties of a median split.

[33] L. Crawford, Senior management perceptions of project management competence, International journal of project management 23 (1) (2005) 7–16.

[34] A. J. Scott, M. Knott, A cluster analysis method for grouping means in the analysis of variance, Biometrics (1974) 507–512.

[35] K. W. Church, Empirical estimates of adaptation: the chance of two noriegas is closer to p/2 than p 2, in: Proceedings of the 18th conference on Computational linguistics-Volume 1, Association for Computational Linguistics, 2000, pp. 180–186.

[36] S. R. Chidamber, D. P. Darcy, C. F. Kemerer, Managerial use of metrics for object-oriented software: An exploratory analysis, Software Engineering, IEEE Transactions on 24 (8) (1998) 629–639.

[37] N. E. Fenton, N. Ohlsson, Quantitative analysis of faults and failures in a complex software system, Software Engineering, IEEE Transactions on 26 (8) (2000) 797–814.

[38] A. G. Koru, H. Liu, Identifying and characterizing change-prone classes in two large-scale open-source products, Journal of Systems and Software 80 (1) (2007) 63–73.

[39] V. Rajlich, A model for change propagation based on graph rewriting, in: Software Maintenance, 1997. Proceedings., International Conference on, IEEE, 1997, pp. 84–91.

[40] P. Ebraert, J. Vallejos, P. Costanza, E. Van Paesschen, T. D'Hondt, Change-oriented software engineering, in: Proceedings of the 2007 international conference on Dynamic languages: in conjunction with the 15th International Smalltalk Joint Conference 2007, ACM, 2007, pp. 3–24.

[41] D. Kung, J. Gao, P. Hsia, F. Wen, Y. Toyoshima, C. Chen, Change impact identification in object oriented software maintenance, in: Software Maintenance, 1994. Proceedings., International Conference on, IEEE, 1994, pp. 202–211.

[42] X. Ren, F. Shah, F. Tip, B. G. Ryder, O. Chesley, Chianti: a tool for change impact analysis of java programs, in: ACM Sigplan Notices, Vol. 39, ACM, 2004, pp. 432–448.

[43] W. Glen, Use value of historical space structures in relation to adaptability for housing, International journal for housing science and its applications 18 (1994) 63–63.

[44] D. M. Gann, J. Barlow, Flexibility in building use: the technical feasibility of converting redundant offices into flats, Construction Management and Economics 14 (1) (1996) 55–66.

[45] G. Bavota, B. Dit, R. Oliveto, M. Di Penta, D. Poshyvanyk, A. De Lucia, An empirical study on the developers' perception of software coupling, in: Proceedings of the 2013 International Conference on Software Engineering, IEEE Press, 2013, pp. 692–701.

[46] D. Poshyvanyk, A. Marcus, R. Ferenc, T. Gyimóthy, Using information retrieval based coupling measures for impact analysis, Empirical software engineering 14 (1) (2009) 5–32.

[47] H. Kagdi, M. Gethers, D. Poshyvanyk, M. L. Collard, Blending conceptual and evolutionary couplings to support change impact analysis in source code, in: Reverse Engineering (WCRE), 2010 17th Working Conference on, IEEE, 2010, pp. 119–128.

[48] A. Lozano, C. Noguera, V. Jonckers, Explaining why methods change together., in: SCAM, 2014, pp. 185–194.

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

[50] H. Abdeen, K. Bali, H. Sahraoui, B. Dufour, Learning dependency-based change impact predictors using independent change histories, Information and Software Technology 67 (2015) 220–235.

[51] M. J. Harrold, P. Kolte, A software metric system for module coupling, Journal of Systems and Software (20) (2003) 295–308.

[52] L. Yu, A. Mishra, S. Ramaswamy, Component co-evolution and component dependency: speculations and verifications, IET software 4 (4) (2010) 252–267.

[53] H. Li, A novel coupling metric for object-oriented software systems, in: Knowledge Acquisition and Modeling Workshop, 2008. KAM Workshop 2008. IEEE International Symposium on, IEEE, 2008, pp. 609–612.

[54] G. A. Hall, W. Tao, J. C. Munson, Measurement and validation of module coupling attributes, Software Quality Journal 13 (3) (2005) 281–296.

[55] M. Gethers, B. Dit, H. Kagdi, D. Poshyvanyk, Integrated impact analysis for managing software changes, in: Software Engineering (ICSE), 2012 34th International Conference on, IEEE, 2012, pp. 430–440.

[56] M. Revelle, M. Gethers, D. Poshyvanyk, Using structural and textual information to capture feature coupling in object-oriented software, Empirical software engineering 16 (6) (2011) 773–811.

[57] L. C. Briand, J. W. Daly, J. K. Wust, A unified framework for coupling measurement in object-oriented systems, Software Engineering, IEEE Transactions on 25 (1) (1999) 91–121.

[58] L. C. Briand, S. Morasca, V. R. Basili, Property-based software engineering measurement, Software Engineering, IEEE Transactions on 22 (1) (1996) 68–86.

[59] S. Morasca, L. C. Briand, Towards a theoretical framework for measuring software attributes, in: Software Metrics Symposium, 1997. Proceedings., Fourth International, IEEE, 1997, pp. 119–126.

[60] F. B. S. Diehl, On the congruence of modularity and code coupling.

[61] N. Hanakawa, Visualization for software evolution based on logical coupling and module coupling, in: Software Engineering Conference, 2007. APSEC 2007. 14th Asia-Pacific, IEEE, 2007, pp. 214–221.

[62] H. Malik, A. E. Hassan, Supporting software evolution using adaptive change propagation heuristics, in: Software Maintenance, 2008. ICSM 2008. IEEE International Conference on, IEEE, 2008, pp. 177–186.

Appendix A. Summary of Structural Coupling Stability Per Project

Table A.6 illustrates the proportion of class dependencies belonging to each of the stability clusters. The 1^{st} column shows the project IDs; the 2^{nd} column shows the number of coupled class pairs; the 3^{rd} column shows the proportion of class pairs that are structurally linked but they never co-change; the 4^{th} column shows proportion of class pairs that are changed in both halves of their life cycle but with a majority of those revisions occurring in the first half of their life cycle; the 5^{th} column shows the proportion of class pairs that equally co-changed in both halves of their structural life cycle but with a majority of the co-changes occurring in the second half; finally in the 6^{th}, the proportion of class pairs with modifications which materialise not when an operational link is established between them but only in a future moment is shown.

Table A.6: Summary of Projects Studied In Terms of Structural Coupling Stability

ID	coupled pairs	% static stable	% stable	% partially stable	% partially unstable	% unstable
1	91	65	24	3	5	1
2	18	5	27	27	33	5
7	65	47	35	1	13	1
8	4082	39	50	2	5	2
10	655	27	40	8	10	13
11	218	27	68	0	3	0
12	118	0	27	13	51	7
13	1662	8	79	3	8	0

Continued on next page

Table A.6 – *Continued from previous page*

ID	coupled pairs	% static stable	% stable	% partially stable	% partially unstable	% unstable
14	1084	51	37	4	4	1
18	67	10	28	37	20	2
20	67	10	28	37	20	2
22	161	21	60	16	1	0
24	317	37	61	0	0	0
26	194	20	48	7	18	4
28	753	40	36	12	8	2
30	157	36	43	7	13	0
31	50	32	36	18	12	2
41	252	5	90	0	2	0
45	13	23	61	7	7	0
51	143	11	72	8	2	4
56	31	74	22	3	0	0
60	674	4	12	3	21	57
64	120	52	35	7	3	0
65	160	51	41	5	1	0
66	2914	25	74	0	0	0
67	76	21	47	23	2	5
68	407	11	85	0	2	0
69	309	21	57	15	3	1
71	476	0	99	0	0	0
79	802	14	72	4	8	0
81	368	47	40	7	2	1
84	659	51	45	0	1	1
86	52	30	69	0	0	0
88	16	0	68	12	18	0
92	259	59	35	1	1	1
96	480	7	15	5	47	23
97	57	45	38	1	14	0
99	1365	38	21	5	12	21
103	80	43	35	5	15	1
107	73	30	68	0	1	0
109	24	16	75	4	4	0

Continued on next page

Table A.6 – *Continued from previous page*

ID	coupled pairs	% static stable	% stable	% partially stable	% partially unstable	% unstable
112	57	19	68	7	5	0
113	55	27	30	18	16	7
115	83	15	28	26	22	6
118	673	26	56	6	9	1
119	23	8	30	52	8	0
122	231	17	48	19	10	3
123	237	40	49	0	8	2
124	43	6	32	30	25	4
127	675	19	57	13	6	2
130	1045	31	60	2	4	0
136	78	5	56	24	11	2
140	127	27	42	9	16	3
141	47	55	31	12	0	0
142	835	18	45	22	10	2
148	189	26	71	0	1	0
149	1526	45	23	5	16	8
152	297	5	67	8	18	0
157	1185	41	41	4	10	1
166	520	65	31	2	0	0
168	1018	50	45	0	3	0
169	50	42	38	4	14	2
170	191	45	34	6	6	7
172	1177	21	24	33	17	2
179	407	33	14	13	17	21
180	367	54	25	1	6	11
183	1457	41	37	9	5	5
184	6594	29	37	2	4	26
185	3954	37	37	11	3	3
186	1341	21	52	7	7	4
188	274	29	38	2	5	12
189	53	3	16	9	20	1
195	376	31	44	5	3	6
197	59	23	20	3	8	30

Continued on next page

Table A.6 – *Continued from previous page*

ID	coupled pairs	% static stable	% stable	% partially stable	% partially unstable	% unstable
201	1652	17	64	2	6	4
202	121	41	36	8	3	3
211	4094	6	58	1	0	0

Appendix B. List of Studied Object-Oriented Software Projects

Table B.7 shows the extent of this issue: the 1^{st} column in Table B.7 shows the project IDs; 2^{nd} column shows the project names; 3^{rd} column shows the number of structural dependencies; 4^{th} column shows the number of logical dependencies; 5^{th} column shows the number of dependencies in the intersection set; 6^{th} further shows the percentage or proportion of structural dependencies in the intersection set; 7^{th} column shows the proportion of logical dependencies in the intersection set. The table is sorted by the 6^{th} column in descending order to depict the observed result outlined in Section 4.1 (a high number of structural dependencies including logical dependencies but not vice versa).

Table B.7: Intersection of Structural and Logical Dependencies in the studied 79 OSS Projects. (KEY: Str. Dep. = Structural Dependencies; Log. Dep. = Logical (change) 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

Continued on next page

Table B.7 – *Continued from previous page*

ID	Project	Str. Dep.	Log. Dep.	Int. Set	CSD	CLD
152	onslaught	297	5,739	289	97	5
166	prettyfaces	519	12,987	500	96	4
96	jbal	480	12,986	461	96	4
124	jutf8search	43	152	41	95	27
115	jprg2-assg	83	332	79	95	24
2	4-connect	18	80	17	94	21
13	alto	1,662	78,481	1567	94	2
211	usemon	4,094	529,590	3,845	94	1
18	apjava	67	196	61	91	31
122	jtowerdefense	231	2,191	210	91	10
68	guavatools	407	6,899	363	89	5
51	echo-nest-java-api	143	1,116	127	89	11
109	jmemcache	24	94	21	88	22
142	monome-pages	835	10,362	727	87	7
79	jangod	802	15,220	697	87	5
127	kryo	675	5,372	580	86	11
112	jnoob	57	417	48	84	12
172	ps3mediaserver	1,177	29,313	983	84	3
26	bitlyj	194	1,036	162	84	16
201	tabulasoftmed	1,652	58,420	1,373	83	2
186	seoma	1,341	16,929	1,104	82	7
20	appletbomberman	282	1,255	230	82	18
28	bluecove	753	63,404	607	81	1
67	gp-net-radius	76	522	61	80	12
69	guitarjava	309	3,681	248	80	7
197	subitizer	59	176	47	80	27
22	ascrblr	161	1,396	128	80	9
118	jroguedps	673	6,255	532	79	9
8	aima-java	4,082	190,432	3,200	78	2
130	lemyriapode	1,045	10,520	809	77	8
165	powermock	2,372	105,733	1,828	77	2
45	dbmigrate	13	26	10	77	38
113	jothelo	55	148	42	76	28
10	alexo-chess	655	9,603	499	76	5

Continued on next page

Table B.7 – *Continued from previous page*

ID	Project	Str. Dep.	Log. Dep.	Int. Set	CSD	CLD
140	mobs	127	672	96	76	14
188	simplenamingservice	274	1,593	205	75	13
11	algmusic	218	3,812	163	75	4
66	gorobot	2914	88,731	2,173	75	2
179	restfb	407	4,045	303	74	7
148	ngamejava	189	1,196	139	74	12
184	semanticdiscoverytoolkit	6,594	177,962	4,741	72	3
107	jiopi	73	532	52	71	10
86	java-weather-api	52	220	37	71	17
195	squabble	376	4,578	267	71	6
31	catchnthrow	50	164	34	68	21
185	semweb4j	3,954	68,309	2,551	65	4
170	projet-qcm-java	191	868	122	64	14
30	castanea	157	624	100	64	16
183	scikit	1,457	10,958	924	63	8
157	p2ploan	1,185	10,041	750	63	7
99	jease	1365	39,842	861	63	2
24	audao	317	6,838	198	62	3
123	jugile-util	237	3,088	144	61	5
7	ahs-scheduling	65	118	39	60	33
169	project-armageddon	50	68	30	60	44
202	tabuvrp–study	121	442	72	60	16
164	powerjava	49	150	29	59	19
103	jeudi-tech-spring	80	310	47	59	15
149	object-procedural-bridge	1,526	27,343	852	56	3
81	jaque	368	1,065	205	56	19
168	product-center	1018	7,220	530	52	7
84	java-chess-web	659	2,596	337	51	13
65	google-voice-java	160	724	81	51	11
14	amock	1,084	2,969	545	50	18
180	robust-coupe	367	1,648	182	50	11
1	2dtetris	91	166	44	48	27
64	geocoder-java	120	379	58	48	15

Continued on next page

Table B.7 – *Continued from previous page*

ID	Project	Str. Dep.	Log. Dep.	Int. Set	CSD	CLD
141	mocrap	47	74	21	45	28
92	javastepbystep	259	1,795	109	42	6
56	fdelimitedtextutilities	31	34	8	26	24

Appendix C. Initial Developer Commit Messages from 10 OSS Projects

Table C.8 shows the commit message found for the initial commit in 10 OSS projects (randomly selected from out pool of 79 studies OSS projects).

Table C.8: Developer Initial Commit Messages of 10 OSS Projects

ID	Project	Initial Commit Message
8	aima-java	"first source checkin"
26	bitlyj	"initial directory structure"
28	bluecove	"Initial import of SourceForge.net Subversion Repository" (first 4 commits)
51	echo-nest-java-api	"initial import"
56	fdelimitedtextutilities	"Initial import of fDTUtils with NetBeans project files. I've been working on this for a couple of days so a lot of the code is in this import"
69	guitarjava	"GameBase: Project created. Initial classes created."
97	jbandwidthlog	no commit message
109	jmemcache	"first version really simple cache management"
152	onslaught	"mavenized project, updated to working version with build and stuff"
172	ps3mediaserver	"first commit"