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

Journal:	*Transactions on Software Engineering*
Manuscript ID	TSE-2016-09-0309
Manuscript Type:	Journal First
Keywords:	D.1.5 Object-Oriented Programming < D.1 Programming Techniques < D Software/Software Engineering, open-source software (OSS), co-evolution, references, coupled logical dependencies (CLD), co-changed structural dependencies (CSD), software classes, association rule mining, structural coupling, logical coupling, co-change

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

Nemitari Ajienka, Andrea Capiluppi

Abstract—During the lifetime of object-Oriented (OO) software systems, new classes which bring about new inter-dependencies are added to increase functionality. Logical coupling depicts the change dependencies between classes, while structural coupling measures source code dependencies induced via the system architecture. In this study, we have analysed 79 open-source software projects of different sizes to investigate the interplay between both types of software dependencies.
Firstly, by statistically computing the correlation between the strengths of logical and structural dependencies. Secondly, by identifying the intersection sets of logically and structurally related class pairs. Thirdly, we propose a way to determine the quality 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 agree with the premise that it is very likely that the structural coupling of classes will lead to them being co-changed at least once in future. However, if classes have been co-changed, it does not necessarily mean that they are structurally coupled. Therefore, co-change could be related to some other software coupling measures (e.g., semantic coupling). We also identified a significant level of structural coupling instability in our overall sample of studied projects.

Index Terms—object-oriented (OO) open-source software (OSS) co-evolution co-change references structural coupling logical coupling software classes association rule mining support and confidence 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, or the inheritance relationships.

Establishing that two software entities *co-evolve* means that developers consider them as logically related: i.e., 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] states 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].

In earlier studies, co-evolution 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 [3], [5], [4].

Figure 2 illustrates what has been proposed in terms of the direction of the relationship between co-evolution and structural coupling for 12 Linux kernel modules [2]. Yu [2] identified a linear and directional relationship between the co-evolution and structural coupling for 12 Linux kernel modules. 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 way and the extent to which changes propagate via structural dependencies are still not clear.

In this context and state of knowledge, this paper analyses a random sample of 79 OSS projects (written in Java) in order to add evidence to the discussion on the causes of co-evolution. This work is based on the three **goals**:

G1: to investigate how the **coupling strength** between classes has an impact on their future co-changes;

G2: to investigate the **directionality** of the relationship between logical and structural coupling; 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 4.5.

Research **questions** were derived from each goal, and testable **hypotheses** formulated for each question, 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

• *Nemitari Ajienka and Andrea Capiluppi are with Brunel University, Uxbridge, Middlesex, UB8 3PH, UK.*
E-mail: nemitari.ajienka,andrea.capiluppi@brunel.ac.uk

TABLE 1: Research Goals and Questions

Goals	Research Questions	Null Hypothesis H_0
G1	[Q1] *Is there a linear relationship between logical and structural coupling?*	No linear relationship between the strengths of logical and structural dependencies
G2	[Q2] *Is there a directional relationship between structural and logical coupling?*	No directional relationship
G3	[Q3] *What is the proportion of stable pairs of classes in a software system?*	The set of stable pairs of classes is larger than the unstable one

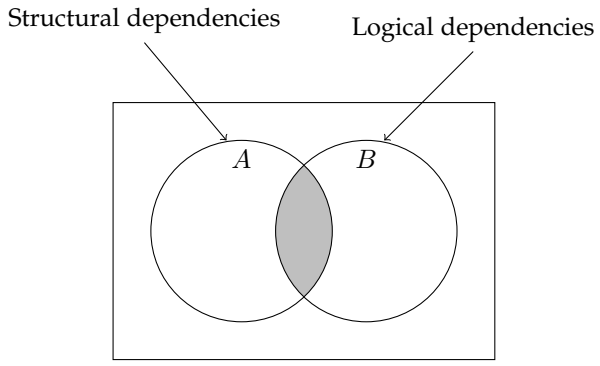

Fig. 1: Intersection of structurally and logically coupled classes

software engineering, including:

- **Prediction of software changes**: Geipel and Schweitzer [4] state that the question about the causes of change propagation has been overlooked by many researches in favor of a predictive approach. As such, these causes are implicitly contained in a prediction function or as input to a machine learning algorithm [6], [8], [9], [10], [11]. A strong relationship between co-evolution and structural coupling provides statistical support for these models and predictions, thus helping to achieve more focused software maintenance.
- **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 structure of a software system.
- **Focusing refactoring effort**: if an analysis of structural coupling reveals a system designed with low coupling between classes; when the analysis of the change history of classes in a software system revealed a high co-evolution between classes, this will be an indication of possible targets for restructuring to decrease unnoticeable coupling between classes in the system [2].
- **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.

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

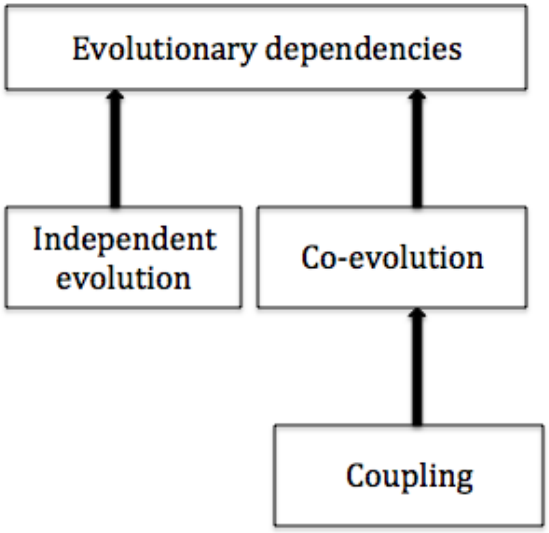

Fig. 2: The relationships among evolutionary dependencies, structural coupling and co-evolution [2] of Linux Kernel Modules.

2 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 the related works on both aspects separately, and when studied jointly.

2.1 Structural Coupling

Structural coupling (called simply "coupling" in some studies [2], [12], [13], [14]) is still considered to be an imprecise measure of software complexity [12]. 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 [15], [16],

change impact analysis [17], [18], [19], [20], re-engineering, reuse, change propagation, and clone management [16].

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) [21], [22], [23].

2.2 Logical Coupling

In comparison to the broad research on structural coupling, the study of logical coupling, evolutionary or change dependencies [2], [3], [6] has just begun a few years ago because of the advances in data mining techniques [2] used to extract co-evolution data. However, despite its short history, there have been several interesting studies published with promising results. Xia [24] argued that the most widely used design metrics for the inter-module relation are based on the information flow rather than the coupling or cohesion criteria, and proposed a metric to compute coupling complexity of modules of a system. Ying *et al.* [8] 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.

2.3 The Link Between Structural and Logical Coupling

For most of the studies described in Subsection 2.1 and 2.2, the study of either structural coupling or co-evolution 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 [25]. To our knowledge, there have only been a few studies that have been performed to understand the relationship between co-evolution and structural coupling [26], [27], [2], [28], [29], [3], [30], [4], [5].

In their seminal study on the Linux co-change, 12 kernel modules (i.e., C files) were studied from the kernel. It was shown that the structural coupling between classes causes them to be co-changed and it plays an important role in the measurement of co-evolution (i.e., structural coupling leads to co-evolution) [2].

Recent studies [3], [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 [17], 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.

Gall *et al.* [29] 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 [29]).

Zimmermann *et al.* [28] 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.* [28] to measure the strength of association rules in our sample. Yu [2] 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 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 the results are replicated with a sample of over 50 Java OSS projects of different sizes and number of revisions in our study though we use a different approach due to the size of our sample.

Fluri *et al.* [30] 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.

Oliva and Gerosa [3] 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, the number of structurally coupled pairs of classes 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 they acknowledge that their results are not really reliable. Thus it is important that this study is conducted using a different sample of projects as well as methodology. In addition, we extracted the source-code references metric between pairs of coupled classes [2] at every snapshot of

each project and use the mode (highest occurring value) to represent the references between the pair of classes in order to deal with the threat of a change in the number of references between classes from one revision to another per software project. [3] also suggests extending their study to other OSS repositories and in this study, the subject systems were taken from the GoogleCode repository.

As opposed to these related studies [3], [30], in addition to quantifying the proportion of logical dependencies that involve non-structurally related elements and vice versa in this study, we also investigate using confidence metrics [28] and the number of references between classes, whether there are evolutionary consequences of structural coupling in a larger sample of OSS projects. In other words, we also identify whether dependencies induced via the system architecture have noticeable effects on the co-evolution of classes or not (independent evolution).

Geipel and Schweitzer [4] analyze the link between structural dependency and co-change. 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 contrast with their study, we extracted the references metric between pairs of coupled classes at every snapshot of each project and use the mode (highest occurring value) to represent the references between the pair of classes in order to deal with the threat of a change in the number of references between classes from one revision to another. 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 [3] and other studies [4], [9], [31], 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.

In a study on 16 OSS projects, using Pearson correlation, Beck and Diehl [26] 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; semantic coupling using the LSI technique 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. 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.

3 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 [3]. In the next Subsections, we introduce structural and logical dependencies and discuss how they can be operationalised in the context of OO programming.

3.1 Logical Coupling

Co-evolution of classes can be represented with their logical or evolutionary dependencies [28], [2] (as shown in Figure 2). However, co-evolution and logical dependency do not mean the same thing.

According to Wiese *et al.*, "*change coupling is a phenomenon associated with recurrent co-changes found in the software history*" [32]. Logical dependencies or evolutionary dependencies are based on the change history of two classes, and is a measurement of the observation that two classes always co-evolve or change together[29], [25], [33], [34]. They are commonly treated as association rules [6], which means that when X_1 is changed, X_2 is also changed [3]. Furthermore, X1 and X2 are called the antecedent (a.k.a, left-hand-side, LHS) and the consequent (a.k.a., 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 [3].

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.* [17] 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.* [35] it is useful to study logical coupling because it can reveal dependencies that are not revealed by analyzing only the source code [2]. This sort of dependencies are the most troublesome and are prone to represent sources of bugs in software projects. In this study we adapt the methods proposed by Zimmermann *et al.* [28] to represent logical dependencies.

3.1.0.1 Operationalisation: The logical dependency between classes, and their strength, is evaluated in this work using the **support** and **confidence** metrics. By doing so, we evaluated the *significance* of the association rules between classes [3], and across the lifespan of a software project (i.e., taking all versions of the software system into consideration).

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 3, class A was modified in 3 transactions (where 3 is the *"Transaction Count"* [2]). 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 $A \leftarrow C$.

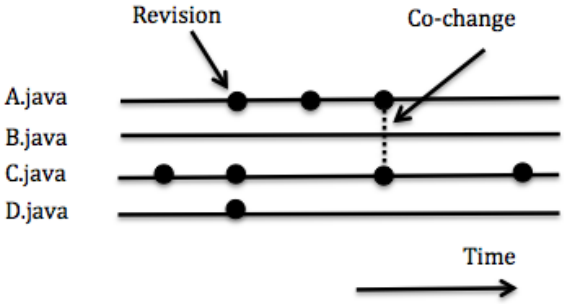

Fig. 3: Association rule example for confidence and support metrics

On the other hand, 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 3, 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.

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 → effect rules can be different.

3.2 Structural Coupling

According to Yu [2], 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. 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 [2] 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 [36]. 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 [37].

3.2.0.1 Operationalisation: In this study, the structural coupling of classes (and its strength) 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 [2] 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" [2]). Accordingly, the strength of the structural coupling from the *caller* C2 to the *called* class C1 in Figure 4 is 4 (2 for function call func(int), 2 for global variable gv) [2].

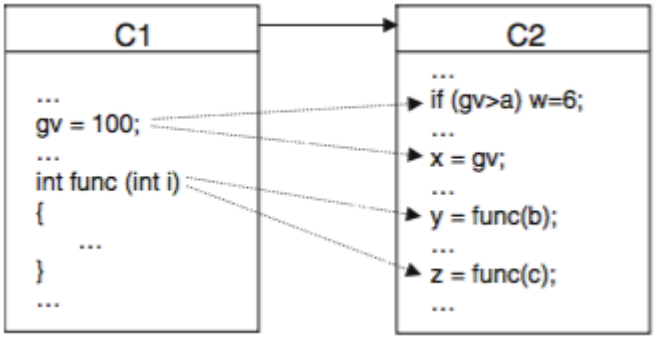

Fig. 4: Structural Dependency Path Between Two OO Software Classes – *Caller* (C2) using a function and variable defined in *Called* (C1) [2]

4 Research Methodology

In this Section, we outline our research methodology with the use of worked examples.

4.1 Case Study

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

1) Medium to large-scale open source projects that (i) provide public access to source code and (ii) use a version control system that allows us to extract the historical information;
2) Projects with multiple revisions/commits (> 20 revisions in order to exclude trivial projects), relatively long history and contributors;

1. Also called the support ratio [2]

3) Projects with a large group of users;
4) Projects implemented in Java to allow the extraction of the structural coupling between classes, since structural coupling varies between languages [3].

4.2 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.

4.2.1 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.

4.2.2 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 process to obtain the metadata for all the projects took around 48 hours: `sleep` statements were inserted in a routine not to overload the online servers, and to make sure that the latest versions of the files were downloaded. The metadata allowed us to obtain the list of revisions for each class, and for the whole project. ~~Table 2~~ summarizes 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 coupling between classes and given that the number of references between two classes could change from revision to revision, the snapshot of the source code was downloaded and stored N times with different IDs, for each project, and according to its revisions: for one project, we ended up downloading its code 769 times (see Table 2, under 'Max'). This phase was one of the most time-intensive in the methodology: to complete the process, and not to overload the online servers, the extraction of the revisions data took over 3 weeks. The sample of projects contains an overall number of 9,269 revisions.

4.3 Identifying class dependencies (RQ1)

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

4.3.1 Logical Coupling

For each project we extracted the number of revisions, based on the tables built by CVSAnalY. This task was a pure SQL extraction task, so it does not pose a time issue. For all revisions, we extracted the list of pairs of classes 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* [4] library in the **R** [5] 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).

4.3.2 Structural Coupling

While logical coupling is based on a time interval, structural coupling is defined for a specific time instant [3], [4]. Each revision 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 [6]. This phase of the methodology was also very laborious, given the vast number of revisions to analyse: with an average of 2 minutes to extract the coupling data of a revision.

After extracting the references metrics between pairs of coupled classes at every snapshot of each project, we computed and used the mode (highest occurring value) to represent the number of references between ~~the~~ any pair of coupled classes per project in order to deal with 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. This process was automated using a Shell script we developed. For example in ~~Table 4,~~ the references between two Java classes (UrSQLController and UrSQLEntity) in the UrSQL project is changed a few times from on revision to another shown in column 2 of Table 4. However, the highest occurring number of references is 15. The idea is that this value should be more representative of the number of references between the classes and is more likely to have the most effect on their co-change or co-evolution.

The structural coupling and co-evolution data, extracted from the sample, are going to be shared in a open repository as well as relevant scripts used to mine the data.[7]

4.4 Evaluating the intersection of sets (RQ2)

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

2. Data dump is available at http://flossdata.syr.edu/data/gc/2012/2012-Nov/
3. http://metricsgrimoire.github.io/CVSAnalY/
4. https://cran.r-project.org/web/packages/arules/index.html
5. https://www.r-project.org/
6. https://scitools.com/
7. http://openscience.us/repo/

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

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

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

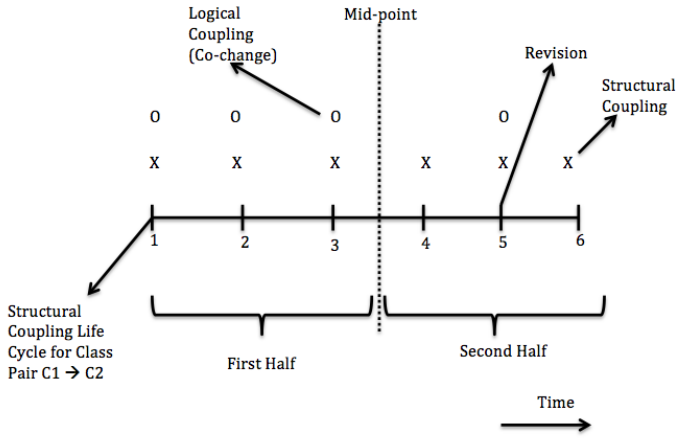

Fig. 5: Evaluating the structural stability for a pair of classes, using structural coupling (x) and co-change (o)

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 (i.e., $A - B$ from the sets in Figure 1), the proportion of logical dependencies that involved non-structural dependencies (i.e., $B - A$ from Figure 1) as well as the intersection set of pairs of classes that are both structurally and logically related (i.e., $A \cap B$ from Figure 1).

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 4) and (iii) assign a level of *stability* to each pair. This was achieved in the following steps:

- we counted the number of revisions when the pair shows a structural link, and we named it the *lifecycle* of that link, and for that pair;
- we divided this lifecycle in two, so to obtain two halves of the lifecycle;
- we identified in which half the pair also showed a co-change;
- we made a decision on the stability of a pair based on patterns of co-change.

This procedure is summarized in Figure 5. The pair C1 → C2 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.

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 [38], [39], adding a coupling link between two classes should ideally only require little or no co-changes or maintenance in both classes in the first half of their coupling life-cycle and not in future if the link is stable or rightly implemented.

4.5.1 Drawing Scenarios of Stability

Using Figure 5 as a visual approach, we identified the following 5 scenarios (also pictorially drawn in ~~Figure 6~~):

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. 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 materialize not when a coupling link is established between them but only in a future moment.

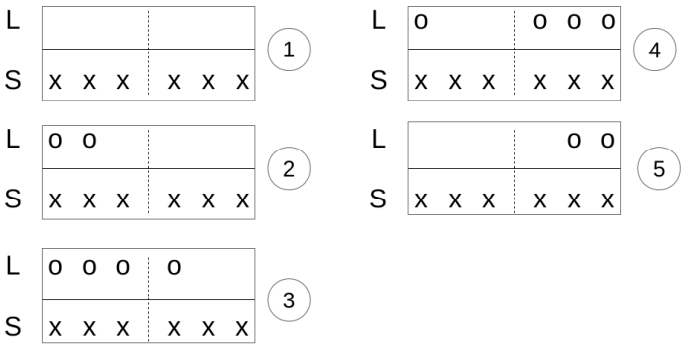

Fig. 6: Summary of the 5 stability scenarios. (KEY: L = Logical Coupling or Co-change; S = Structural Coupling)

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 whenever there is a new requirement [1].

4.6 Statistical tests – Spearman's Correlation (RQ1)

This Section describes the computation of statistical tests for RQ1. The intersection of coupling sets (from 4.4) 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 and along their string of revisions. 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 dependencies.*

The correlation between the two vectors is evaluated using the Spearman's rank correlation coefficient [2]. Spearman's metric was chosen because it is unlikely that either the structural or logical coupling values will have a normal distribution. Additionally, some classes might not be changed in all the revisions in which they are structurally coupled, so the two vectors can have different size.

We reject the null hypothesis for all the projects studied 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 5.

The $\alpha = 0.05$ level was chosen as suggested in Yu's study [2]. One of the threats to the statistical validity to their study was the selection of the significance level. In that study, they chose $\alpha = 0.1$ which might have resulted in a type I error – mistakenly rejecting a null hypothesis. To reduce this threat they planned to in the future research, increase the α value to 0.05 for more accuracy which we have done in this study.

5 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 research questions outlined in Table 1.

5.1 RQ1. *Is there a «linear» relationship between structural and logical coupling?*

To answer this research question, we used the method outlined in Section 4.3 and the statistical approach shown in Section 4.6. Using the Spearman's rank correlation, we tested for the null hypothesis H_0: *There is no linear relationship between the logical and structural coupling of OO software classes* (at $\alpha = 0.05$).

Differently from [2], 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 support H_0. Using Spearman's correlation (with $\alpha <= 0.05$) and in all the projects analysed, we reject H_0: there is not a correlation or linear relationship between the strength of the structural and logical coupling of classes in OO software systems. The results of the tests are visible in Figures 7 and 8 below.

Figure 7 shows the generic correlation outcomes (using the Pearson product moment correlation coefficients), alongside the p-values derived from the Spearman's rank correlation analysis computation. As visible, there is a correlation (i.e., p-value <= 0.05) for only 12 projects. In a handful of projects we observed a negative Pearson'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 a positive Pearson's correlation coefficient.

Figure 8 shows the overall distribution of the Spearman's p-values, in form of a box-plot. Considering both Figures 7 and 8, for the majority of projects there is no correlation or linear relationship between the strengths of the logical and structural dependencies in OO software projects. The outcome of this test was concluded considering the overall pool of projects, rather than the single projects alone.

5.2 RQ2. *Is there a «directional» relationship between structural and logical coupling?*

To answer this question, the aim was to get a view of the intersection set of pairs of structurally and logically related classes in OO software projects. Once the two sets of coupling are computed per project, the shaded region of the

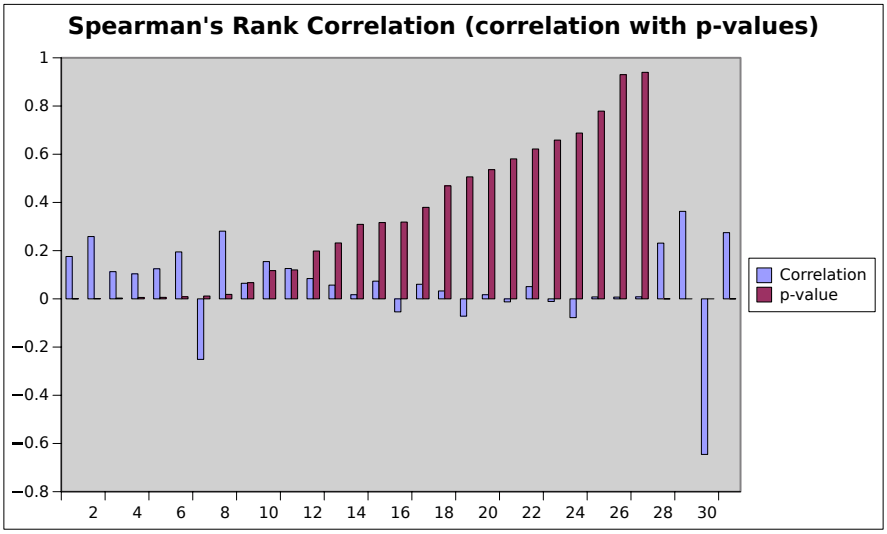

Fig. 7: RQ1 - Spearman's Rank Correlation (correlation results with p-values)

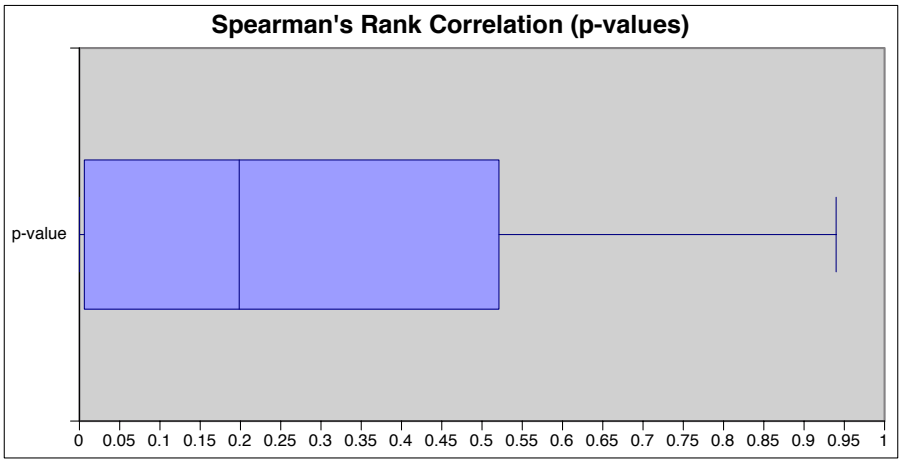

Fig. 8: RQ1 - Spearman's Rank Correlation (box-plot distribution of p-values)

Venn diagram ($A \cap B$) in Figure 1 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. ~~Table 6~~ in the Appendix shows the extent of this issue: the 1^{st} column in Table 6 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.

From Table 6 we know, for example, that the project with ID=31 has a 68% of its coupled pairs that also co-changed at some point. On the other hand, only 21% of the pairs that co-changed are also structurally coupled, 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. Overall, 64 out of the 79 projects show that structural coupling leads to co-change: between 60 and 100% of all the structurally coupled pairs of classes co-changed at least once in the evolution of the projects.

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. The proportion of logically related classes that involve structurally related classes OO software is very low in all projects studied.

This confirms the directional relationship (structural coupling $\rightarrow$ co-evolution) between structural and logical coupling, as identified by Yu [2] and shown in Figure 2. Therefore, we reject the null hypothesis (H_0) but fail to reject the alternative hypothesis (H_1); *There is a directional relationship between structural and logical coupling in OO software.*

The proportion of structurally related classes that involve logically related classes in OO software is quite high in a majority (81%) of the projects. In a few projects, **all** the structural dependencies are reflected into logical dependencies. In both venn diagrams (top and bottom) in Figure 9, the smaller circle represents the set of structural dependencies while the larger circle represent the set of logical dependencies.

Using the top (weighted) Venn diagram in Figure 9,

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.

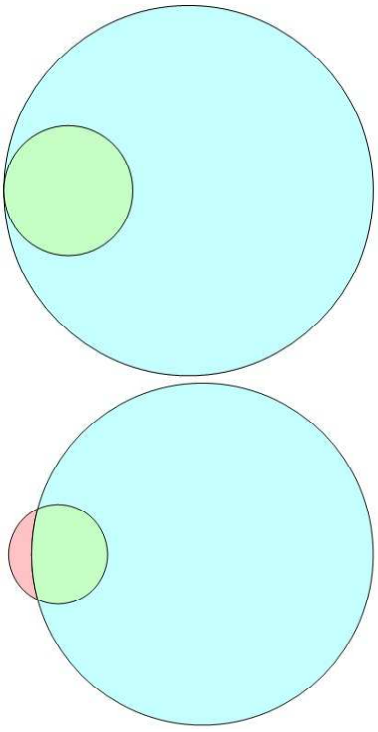

Fig. 9: Venn Diagrams (weighted) showing the two sets of coupling in two scenarios: project ID=97 (top) and project ID=69 (bottom)

The second most common scenario identified in the results is illustrated using the bottom venn diagram in Figure 9 (bottom), 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.

Figure 10 shows two summary box-plots with the following percentages:

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

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

The Co-changed Structural Dependencies ratio (CSD) is the percentage of structurally coupled pairs that are also co-changing with respect to all the structurally coupled pairs. *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 is a similar percentage, but evaluated with the co-change sets. The two boxplots 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.

Combined with the results from RQ1, in a summary we observed that structural coupling does cause co-change in the majority of projects. What we were not able to sustain is correlation between the two types of coupling. The strength of structural coupling between two classes is not a factor in the successive co-changes of those classes.

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

To answer RQ3, we evaluate the overall stability of the studied sample of projects by using the five stability scenarios from Section 4.5.

Using the *2dTetris* project (ID=1) as an example, the scenarios of its structural stability are summarized in ~~Figure 11.~~ It is evident that the project has a large proportion (85%) of stable pairs (as the sum of the hatched bars), and a lesser proportion of unstable pairs of classes (sum of the red bars). This analysis clearly points out the unstable pairs of classes that, in the *2dTetris* project, keep being co-changed long after they have being introduced into the system. This by itself is already a cause of additional maintenance costs, that ideally should be kept low during development.

Repeating the same analysis for all the projects, Appendix B shows the percentages of pairs falling in each scenarios, per project (Table 5). Figure 12 further summarizes the overall sample of 79 studied OO software projects. Boxplots are used for the 5 scenarios.

Looking at the box-plots:

- several projects have a relatively large ratio of unstable pairs. One project in particular (*hobbylinkchecker*) contains 98% 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 9% and 14%, respectively, on average).
- The percentages of stable and static-stable pairs show the higher variability, given the sample.
- Only 10 projects show a proportion of "unstable" and "partially stable" that is larger than the sum of the stable clusters. All the other projects have much more stable pairs than unstable.

Our results suggest that the null hypothesis for RQ3 shown cannot be rejected: *The majority of systems in the sample show a larger set of stable pairs of classes than unstable*. We instead reject the alternative hypothesis H_1 (*The majority of systems in the sample do not show a larger set of stable pairs of classes than unstable*).

6 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 [2] and (iii) overall evaluation of the structural stability of OO software projects.

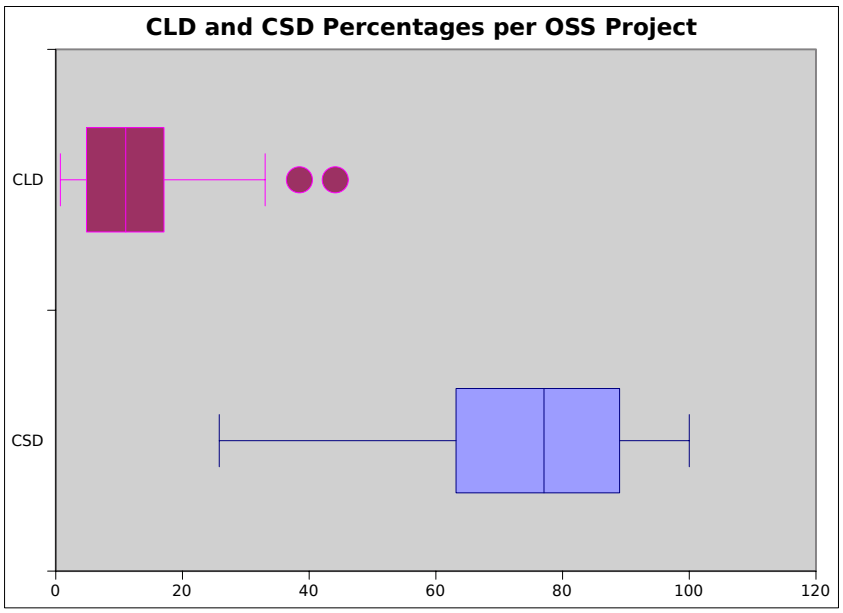

Fig. 10: CLD and CSD Percentages per OSS Project (KEY: CSD = Co-changed Structural Dependencies; CLD = Coupled Logical Dependencies)

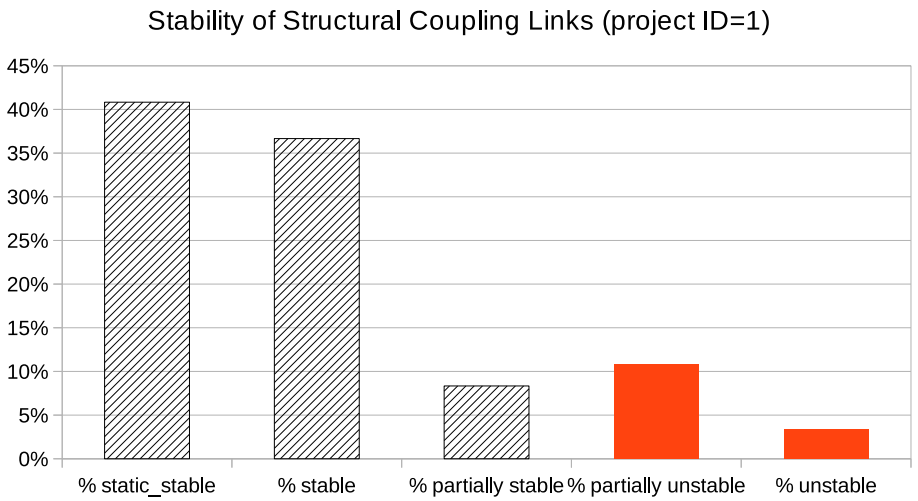

Fig. 11: Stability of Structural Coupling Links in 2dTetris (Project ID = 1)

Below we discuss the findings reported, and put them in perspective for the software maintenance field.

6.1 Linear relationship between the structural and logical coupling of OO software classes – RQ1

The results of the analysis of RQ1 are presented in Section 5. When using $\alpha <= 0.1$, [2] 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.

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 13 below.

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 orientated to co-changes.

6.2 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

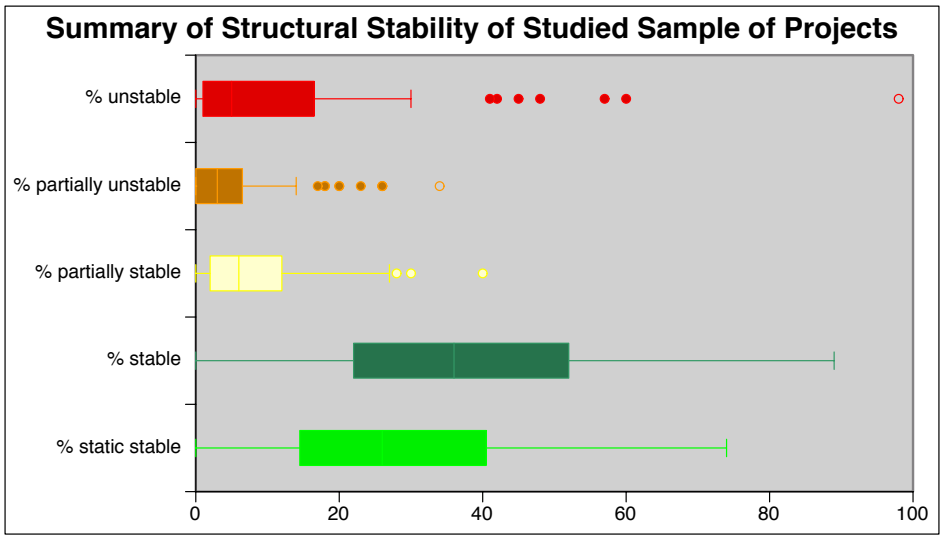

Fig. 12: Summary of Structural Stability of Studied Sample of Projects

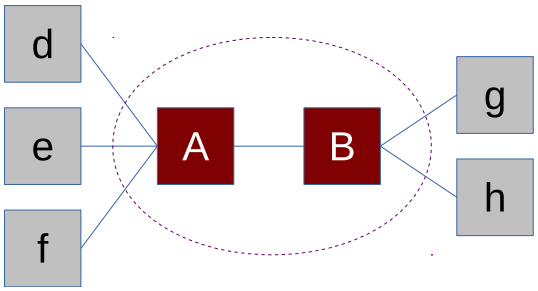

Fig. 13: Effect of networks

larger than the proportion of coupled logical dependencies (CLD) in open-source software projects.

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 artefacts (complexity [40], defect density [41], number of changes [42]), we have detected that pairs of structurally coupled classes do not follow such distributions.

The structural coupling between two classes is therefore a strong predictor of future unplanned and necessary co-changes. On the other hand, the logical coupling does represents a much larger set of interaction between pairs of classes, and does not assist in highlighting the classes that are structurally linked.

6.3 Object-Oriented Software Structural Stability

Having posed a direction to the structural -> logical coupling relationship, we finally investigated how time affects a structural link between classes. We posit that after insterting structural coupling 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 [38] 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 [38] 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 [43], [44], [45]), (ii) capacity (load intake, volume) and (iii) flow, or the movement within buildings (i.e., change in the access scope and mode for any datum in software systems [45], [46]). Finally, the *nature* of component interactions influence the flexibility of building structures to the different maintenance types [38], [47], [48].

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.

6.4 Discussion Summary

Geipel and Schweitzer rightly state that any model that tries to infer structural coupling from co-change will produce a lot of false positives [4]. On the other hand, using the structural coupling information between pairs of classes to predict unplanned future co-changes is a more realistic objective [5].

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 [49].

According to Bavota *et al.* [16] "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 [18] and have combined semantic and logical coupling metrics in change impact analysis [50], [51]. However, there is still the need to study the interplay between semantic and logical coupling in OO software [3], [5].

7 Threats To Validity

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

7.0.0.1 External validity: This paper presents the results of an empirical analysis that should be applicable to all OSS projects. We cannot generalise 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.

7.0.0.2 Internal validity: We acknowledge the fact that support and confidence values of association rules could produce misleading results [3]. 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 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.

7.0.0.3 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.

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, 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 two types of software dependencies.

Secondly, 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 absence of a bi-directional relationship, but the presence of a one-way directional relationship between structural and logical coupling (structural $\rightarrow$ logical) in a majority of the software projects.

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.

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).

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

[3] 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.

[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] 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.

[9] 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.

[10] 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.

[11] 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.

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

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

[14] 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.

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

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

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

[23] 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.

[24] F. Xia, Module coupling: A design metric, in: Software Engineering Conference, 1996. Proceedings., 1996 Asia-Pacific, IEEE, 1996, pp. 44–54.

[25] 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.

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

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

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

[37] 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.

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

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

[40] 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.

[41] 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.

[42] 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.

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

[44] 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.

[45] 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.

[46] 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.

[47] 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.

[48] 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.

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

[50] 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.

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

APPENDIX

TABLE 5: Summary of Projects Studied In Terms of Structural Coupling Stability

ID	coupled pairs	% static stable	% stable	% partially stable	% partially unstable	% unstable
1	91	51	17	6	10	7
2	18	5	27	27	5	5
7	65	47	20	7	0	18
8	4082	39	22	4	9	7
10	655	27	18	6	26	2
11	218	27	69	1	0	0
12	118	0	25	13	20	9
13	1662	8	78	3	1	2
14	1084	51	37	4	1	1
18	67	10	19	40	5	2
20	282	20	62	13	0	0
22	161	21	60	16	0	0
24	317	37	1	0	0	60
26	194	20	31	8	5	19
28	753	40	38	9	3	2
30	157	36	43	7	2	0
31	50	32	38	8	8	0
41	252	5	89	0	0	0
45	13	23	38	7	0	23
51	143	11	49	6	4	12
56	31	74	22	3	0	0
60	674	4	12	3	17	57
64	120	52	16	3	3	20
65	160	51	27	1	0	18
66	2914	25	74	0	0	0
67	76	21	39	30	1	3
68	407	11	32	2	2	45
69	309	21	52	14	5	1
71	476	0	0	0	0	98
79	802	14	72	4	4	0
81	368	47	13	5	11	18
84	659	51	23	1	5	12
86	52	30	26	0	0	42
88	16	0	68	12	18	0
92	259	59	30	1	0	5
96	480	7	5	6	34	41
97	57	3	45	12	26	1
99	1365	38	5	0	4	48
103	80	43	43	3	0	1
107	73	30	68	0	0	0
109	24	16	62	4	0	12
112	57	19	63	8	0	3
113	55	27	18	16	14	7
115	83	15	28	26	9	6
118	673	26	56	6	5	1
119	23	8	30	26	26	0
122	231	17	40	20	7	6
123	237	40	47	0	2	3
124	43	6	32	27	6	4
127	675	19	52	14	2	5
130	1045	31	59	3	1	0

Continued on next page

TABLE 5 – *Continued from previous page*

ID	coupled pairs	% static stable	% stable	% partially stable	% partially unstable	% unstable
136	78	5	53	28	0	7
140	127	27	31	11	3	11
141	47	55	31	12	0	0
142	835	18	44	21	4	3
148	189	26	64	0	0	6
149	1526	45	17	2	6	16
152	297	5	62	6	2	5
157	1185	41	20	3	3	19
164	49	44	30	12	2	0
165	2372	26	36	3	7	17
166	519	5	31	10	23	21
168	1018	50	33	0	0	9
169	50	42	38	4	4	4
170	191	45	36	5	1	6
172	1177	21	15	25	18	5
179	407	33	15	18	4	20
180	367	54	22	1	1	18
183	1457	41	35	7	3	6
184	6594	29	37	2	0	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
201	1652	17	64	2	6	4
202	121	41	36	8	3	3
211	4094	6	58	1	0	0

TABLE 6: 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	35923	473	99	1
41	daedalum	252	4854	249	99	5
136	migrator-postgresql	78	476	76	97	16
60	fyllgen	674	14318	656	97	5
152	onslaught	297	5739	289	97	5
166	prettyfaces	519	12987	500	96	4
96	jbal	480	12986	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	1662	78481	1567	94	2
211	usemon	4094	529590	3845	94	1

Continued on next page

TABLE 6 – *Continued from previous page*

ID	Project	Str. Dep.	Log. Dep.	Int. Set	CSD	CLD
18	apjava	67	196	61	91	31
122	jtowerdefense	231	2191	210	91	10
68	guavatools	407	6899	363	89	5
51	echo-nest-java-api	143	1116	127	89	11
109	jmemcache	24	94	21	88	22
142	monome-pages	835	10362	727	87	7
79	jangod	802	15220	697	87	5
127	kryo	675	5372	580	86	11
112	jnoob	57	417	48	84	12
172	ps3mediaserver	1177	29313	983	84	3
26	bitlyj	194	1036	162	84	16
201	tabulasoftmed	1652	58420	1373	83	2
186	seoma	1341	16929	1104	82	7
20	appletbomberman	282	1255	230	82	18
28	bluecove	753	63404	607	81	1
67	gp-net-radius	76	522	61	80	12
69	guitarjava	309	3681	248	80	7
197	subitizer	59	176	47	80	27
22	ascrblr	161	1396	128	80	9
118	jroguedps	673	6255	532	79	9
8	aima-java	4082	190432	3200	78	2
130	lemyriapode	1045	10520	809	77	8
165	powermock	2372	105733	1828	77	2
45	dbmigrate	13	26	10	77	38
113	jothelo	55	148	42	76	28
10	alexo-chess	655	9603	499	76	5
140	mobs	127	672	96	76	14
188	simplenamingservice	274	1593	205	75	13
11	algmusic	218	3812	163	75	4
66	gorobot	2914	88731	2173	75	2
179	restfb	407	4045	303	74	7
148	ngamejava	189	1196	139	74	12
184	semanticdiscoverytoolkit	6594	177962	4741	72	3
107	jiopi	73	532	52	71	10
86	java-weather-api	52	220	37	71	17
195	squabble	376	4578	267	71	6
31	catchnthrow	50	164	34	68	21
185	semweb4j	3954	68309	2551	65	4
170	projet-qcm-java	191	868	122	64	14
30	castanea	157	624	100	64	16
183	scikit	1457	10958	924	63	8
157	p2ploan	1185	10041	750	63	7
99	jease	1365	39842	861	63	2
24	audao	317	6838	198	62	3
123	jugile-util	237	3088	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	1526	27343	852	56	3
81	jaque	368	1065	205	56	19
168	product-center	1018	7220	530	52	7
84	java-chess-web	659	2596	337	51	13
65	google-voice-java	160	724	81	51	11
14	amock	1084	2969	545	50	18

Continued on next page

TABLE 6 – *Continued from previous page*

ID	Project	Str. Dep.	Log. Dep.	Int. Set	CSD	CLD
180	robust-coupe	367	1648	182	50	11
1	2dtetris	91	166	44	48	27
64	geocoder-java	120	379	58	48	15
141	mocrap	47	74	21	45	28
92	javastepbystep	259	1795	109	42	6
56	fdelimitedtextutilities	31	34	8	26	24