Experience Report: How do Structural Dependencies Influence Change Propagation? An Empirical Study

Gustavo Ansaldi Oliva, Marco Aurélio Gerosa
Department of Computer Science
University of São Paulo (USP)
São Paulo, Brazil
{goliva,gerosa}@ime.usp.br

Abstract**—** Real world object-oriented systems are composed of hundreds or even thousands of classes that are structurally interconnected in many different ways. In this highly complex scenario, it is unclear how changes propagate. Given the high maintenance cost brought by change propagation, these questions become particularly relevant in practice. In this paper, we set out to investigate the influence of structural dependencies on change propagation. We historically analyzed thousands of code snapshots coming from 4 open-source Java projects of different sizes and domains. Our results indicated that, in general, it is more likely that two 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 we track down dependencies 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. Finally, we also found several cases where software changes could not be justified using structural dependencies, meaning that co-changes might be induced by other subtler kinds of relationships.

Keywords— change propagation; structural dependencies; dependency analysis; mining software repositories; software analysis; software maintenance

I. Introduction

"Avoid strong coupling." This has been a mantra in Software Engineering since its early days [20]. The consensus is that strong coupling is undesired because software modules might undergo forced local changes due to changes in related modules [8, 20, 21]. The hazardous effects of change propagation to software maintenance have also been discussed in seminal works [4, 10, 16].

However, controlling coupling levels in practice is still challenging. One of the reasons has to do with the fact that the way and the extent to which changes propagate via structural dependencies is still not clear. To make things even more complicated, there are several kinds of structural dependencies and several ways the changes can happen. This all makes it difficult to focus dependency management and thus to prevent large chains of change propagation.

A concrete example. We mined Apache Lucene v5.1.0 and discovered that its core of has 687 compilation units (Java files), and around 25% of them are connected to at least 8 other units. Furthermore, the same proportion of compilation units span at least 31 structural dependencies. From a change propagation perspective, should developers care about all of these dependencies? Do changes propagate through them? How often? What is the role of the change itself and its inherent context?

Building on our previous studies [13] and on recent work in the field [6, 7, 9], we set out to empirically investigate the influence of structural dependencies on change propagation. We focus on a quantitative analysis of how these dependencies relate to the occurrence of co-changes found in the version history of the subject systems. We performed our analysis on 4 open source Java projects, namely Apache Lucene, Apache Tomcat 7, Megamek, and Apache Commons CSV. We answered the following research questions:

RQ1: Are dependent files more likely subject to co-change than independent ones?

Given a pair of Java files <f_1,f_2>, our results indicated that, even though it is more likely that f_1 and f_2 *will not co-change* just because f_1 depends on f_2 (i.e., dependencies do not instantly make two files change together), the rate with which f_1 co-changes with f_2 is higher when f_1 structurally depends on f_2 (as compared to when f_1 does not depend on f_2). However, this rate is fairly low, with its median ranging from 13.5% (Commons CSV) to 20% (Lucene).

RQ2: Are dependent files more likely subject to co-change than independent ones when the change context is taken into account?

In this research question, we take the change context into consideration. In this new scenario, we consider that there is a dependency from f_1 to f_2 only if f_1 depends on "something" that changed in f_2 (e.g., a field definition or a method's body). The results vary substantially compared to those we obtained in RQ1. Given a pair of Java files <f_1,f_2>, when f_1 depends on f_2 and f_2 changes, it is more likely that f_1 *will co-change* with f_2 in Lucene and Tomcat. In Commons CSV, the number of co-changes and absence of co-changes is roughly the same in the presence of structural dependencies. In Megamek, it is more likely f_1 *will not* co-change with f_2.

Furthermore, we also found that the rate with which f_1 co-changes with f_2 is again higher when f_1 structurally depends on f_2 (as compared to when f_1 does not depend on f_2). This rate is also substantially higher than that we found when ignoring the change context (RQ1), ranging from 43.08% (Commons CSV) to 100% (Lucene and Tomcat).

RQ3: Do changes propagate via structural dependencies?

We focused on an analysis of the proportion of changed methods that propagated changes via call dependencies to other methods per commit. The distribution of this proportion was similar in all projects, with a median of zero. The mean varied from 11.40% (Tomcat) to 18.55% (Lucene). Hence, we concluded that changes in methods rarely propagate via call dependencies. This result supports the idea found in some related work that few changes are "justified" by the software architecture [6].

The key take-home message of this paper is that the relationship between structural dependencies and software changes are not as straightforward as one might think. The results of RQ1 and RQ2 reiterate the importance of managing structural dependencies while evolving software systems. This calls for advanced supporting tools that are capable of, for example, measuring the levels of change propagation over time and alerting end-users when thresholds are exceeded. Commercial tools used in industry like IBM RSA and Stan4J only act on a code snapshot and thus are not able to offer features like this. In turn, the results of RQ3 show that the relationship between individual software changes made in a commit might very loosely related to the software architecture. This reinforces the need of researching the relationship between change propagation and the many forms of connascence [15], such as conceptual coupling [17].

Paper Organization. The rest of this paper is organized as follows. In Section II, we present the theoretical background of this study. In Section III, we describe the design of this study, which includes explaining our goals, the procedures for subject systems selection and data collection, and the tools we used. In Section IV, present the detailed results for the three aforementioned research questions. In Section V, we discuss related work. In Section VI, we discuss the threats to the validity of this study. Finally, in Section VII, we state our conclusions and plans for future work.

II. BACKGROUND

A. Terminology for Java entities

In order to avoid possible misunderstandings due to the plurality of terms and meanings in Software Engineering, we show the terminology we will employ throughout this paper to refer to Java entities[1]. We focus on the Java terminology because all subject systems evaluated in this study were written in Java.

A **compilation unit** is the smallest unit of source code that can be compiled. In the current implementation of the Java platform, the compilation unit is file (often with the .java extension). A **type** is a class or interface. If type X *extends* or *implements* type Y, then X is a **subtype** of Y. We also say that Y is a **supertype** of X. A **class** is a type that defines the implementation of a particular kind of object. A class definition defines instance and class variables and methods, as well as specifying the interfaces the class implements and the immediate superclass of the class. If the superclass is not explicitly specified, the superclass will implicitly be `Object`. An **interface** is a collection of method definitions and constant values. An interface can be implement by one or more classes. **Type Members** are the attributes and methods defined within the context of a type.

B. Structural Dependencies

According to the Merriam-Webster dictionary, a dependency refers to "the quality or state of being dependent; especially: the quality or state of being influenced or determined by or subject to another." Software systems are composed of artifacts that depend on one another during design time and runtime. These dependencies connecting software artifacts are vaguely known as *software dependencies*.

Structural dependencies, also known as *syntactic dependencies*, are a particular kind of software dependency. In the context of object-oriented programming languages like C++, C#, and Java, *structural dependencies* occur whenever a compilation unit depends on another at either compilation or linkage time.

There are several kinds of structural dependencies. This study takes most of them into account (Figure 1). **Member-to-member dependencies** are dependencies where both client and supplier are type members. It is further specialized into two kinds. **Access dependency** refers to a type's attribute access within the body of a certain method. A **method call dependency** refers to a constructor call while defining an attribute or a method invocation (including constructor call) inside a method's body.

Member-to-type dependencies are dependencies where the client is a method and the supplier is a type. It is further specialized into four kinds. **Parameter dependency** refers to an input parameter of some type as part of the method signature definition. A **reference dependency** denotes a reference to a type while defining an attribute or a reference to a type inside a method's body. A **return dependency** refers to the return type of a method declaration. A **throw dependency** refers to the exception types thrown as part of the method declaration.

Type-to-type dependencies are dependencies where both client and supplier are types. It is further specialized into three kinds. **Class inheritance dependency** occurs when a class extends another. **Interface inheritance dependency** occurs when an interface extends another. An **interface implementation dependency** occurs when a class implements an interface.

Import dependencies are dependencies where the client is a compilation unit and the supplier can be either a type or a method (in the case of static imports). It is further specialized into two kinds. A **type import dependency** refers to a type import declaration as part of the compilation unit header. A **method import dependency** refers to a method static import declaration as part of the compilation unit header.

[1] Extracted from https://docs.oracle.com/javase/tutorial/information/glossary.html

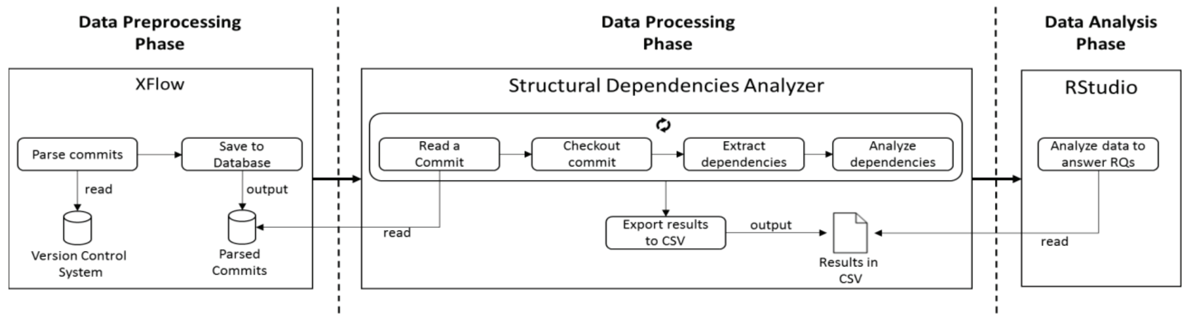
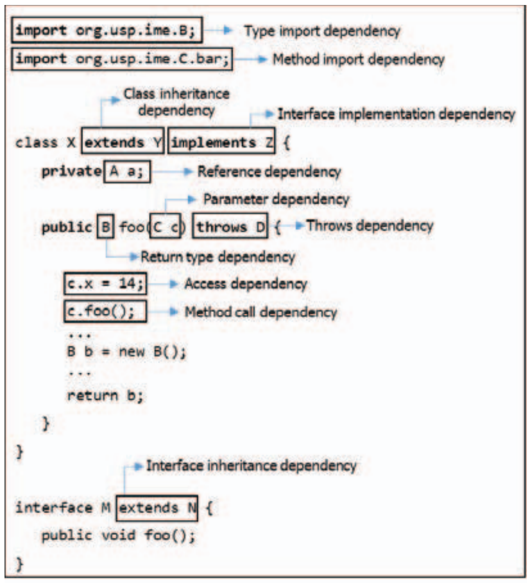


Fig. 2. Main steps of the study design



```
import org.usp.ime.B;
import org.usp.ime.C.bar;
class X extends Y implements Z {
   private A a;
   public B foo(C c) throws D {
      c.x = 14;
      c.foo();
      ...
      B b = new B();
      ...
      return b;
   }
}
interface M extends N {
   public void foo();
}
```

Fig. 1. Structural dependencies exemplified in a sample Java code exceprt.

III. Study Design

A. Goals and Research Questions

The *goal* of this study is to empirically determine the influence of structural dependencies on change propagation. The *focus* is on the evolution of open-source Java applications. Our *perspective* is that of developers, as they need to be aware of and manage coupling while evolving software systems.

Our research questions are the following. **RQ1:** Are dependent files more likely subject to co-change than independent ones? **RQ2:** Are dependent files more likely subject to co-change than independent ones when the change context is taken into account? **RQ3:** Do changes propagate via structural dependencies?

B. Selection of Subject Systems

We pre-established a set of requirements for choosing the subject projects. First, as the identification of structural dependencies is programming language dependent, we decided to pick projects written in the same language. We chose Java, since there are many open0source projects written in this language and we had an appropriate tool to extract Java dependencies. This decision also enabled us to compare the results across projects. Second, due to limitations of our tool set, we chose projects versioned with Subversion (SVN) only. Third, in order to improve the external validity of our findings, we selected projects of different sizes and domains. Finally, since we want to investigate the link between structural dependencies and co-changes, we need to extract these dependencies in every code snapshot of the subject system. As this is a computationally-intensive task, we limited our analysis to 4 systems.

C. Data Collection

All Apache Software Foundation projects in Subversion are hosted under the same repository. As of August 24, 2015, this repository hosts more than 1.69 million commits. Working with large remote repositories poses a series of challenges. To cope with them, we built a local mirror of the whole repository. In fact, we already had this mirror in hands, as it was built for previous studies we conducted [12, 14].

D. Data Preprocessing

We used XFlow to collect, parse, and store commit metadata (Figure 2 – Data Preprocessing Phase). We relied on such data to determine which commits to analyze, as we are only interested in commits that included Java files. We also relied on these data to gain more insight about the subject systems, including their age, number of developers, and number of Java files per commit.

E. Data Processing and Analysis

In this section, we detail the data processing and analysis phases of the study design. As depicted in Figure 2, for each parsed commit, we read it, check-out the corresponding code snapshot, extract dependencies, and analyze them according to the specific research question at hand. After all code snapshots are analyzed, we export the results to a CSV file and analyze it using RStudio. More details about the tools we used in this study are given in Section III.F.

In the following, we discuss the dependency analysis method we employed to answer each of three research questions we raised in this paper.

1) RQ1 – Data Processing and Analysis

To address this research question, we mined the version control system to capture dependencies and co-changes (algorithm I describes the steps we followed in details). For each commit, the corresponding code is checked-out. Afterwards, each file f_2 in the commit is compared with each file f_1 in the checked-out code in order to discover two things: whether f_1 is a client of f_2 (line 07) and whether they co-changed (line 08). A file f_1 is a client of f_2 if there is at least one dependency from f_1 to f_2 (e.g., when a certain method from f_1 calls a method defined in f_2). The kinds of dependencies we capture are described in Section II.B. We also say that a file f_1 co-changes with f_2 when both are included in the same commit.

A `rel` (relationship) object stores how many times f_1 depended (and did not depend) on f_2, as well as how many times f_1 co-changed (and did not co-change) with f_2. This object is kept in memory and retrieved on demand (line 05). The `rel` object for f_1 and f_2 is updated every time f_2 is included in a commit and f_1 is in the checked-out code.

Algorithm I: Recording co-changes and structural dependencies

```
01. for each commit c do
02.   code <- checkout(c)
03.   for each file f₂ in commit c do
04.     for each file f₁ ≠ f₂ in the code do
05.       rel <- getRelationship(f₁,f₂)
06.       if(rel is null) createRel(f₁,f₂)
07.       isClient <- is f₁ a client of f₂?
08.       hasCoChanged <- does c contain f₁?
09.       rel.update(isClient,hasCoChanged)
```

Listing 1. Mining algorithm that tracks co-changes and dependencies.

After mining all commits, we obtained the data shown in Table I for every single pair of files evaluated. From such data, we derived a series of metrics:

TABLE I. RELATIONSHIP RESULTS

File pair $<f_1,f_2>$		Is f_1 a structural client of f_2?	
		Yes (there is at least one dep. from f_1 to f_2)	**No** (there are no deps. from f_1 to f_2)
Does f_1 co-change with f_2?	**Yes** (f_2 changes and f_1 changes as well)	CD	$C\bar{D}$
	No (f_2 changes and f_1 does not change)	$\bar{C}D$	$\bar{C}\bar{D}$

- `CoChanges(f₁,f₂)`: Number of commits in which f_1 co-changed with f_2: $CD + C\bar{D}$
- `NoCoChanges(f₁,f₂)`: Number of commits in which f_1 *did not* co-change with f_2: $\bar{C}D + \bar{C}\bar{D}$
- `Dep(f₁,f₂)`: Number of commits that included f_2 and f_1 depended on f_2: $CD + \bar{C}D$
- `NoDep(f₁,f₂)`: Number of commits that included f_2 and f_1 *did not* depend on f_2: $C\bar{D} + \bar{C}\bar{D}$
- `CoChangeRatioWithDep(f₁,f₂)`: Of all commits that included f_2 and f_1 depended on f_2, how many times f_1 co-changed with f_2? $\frac{CD}{CD+\bar{C}D} = \frac{CD}{Dep(f_1,f_2)}$
- `CoChangeRatioWithoutDep(f₁,f₂)`: Of all commits that included f_2 and f_1 *did not* depend on f_2, how many times f_1 co-changed with f_2? $\frac{C\bar{D}}{C\bar{D}+\bar{C}\bar{D}} = \frac{C\bar{D}}{NoDep(f_1,f_2)}$

These metrics were exported to a CSV file and then statistically analyzed in RStudio (Figure 2). As a preliminary analysis, we compared the distributions of CD and $\bar{C}D$ to understand the frequency of co-changes in the presence of structural dependencies. Next, to answer the research question, we investigated whether files with structural dependencies are more likely to co-change. We accomplished that by checking whether the rate with which f_1 co-changes with f_2 is higher when f_1 structurally depends on f_2 (as compared to when f_1 does not depend on f_2). That is, we expect `CoChangeRatioWithDep(f₁,f₂)` to be higher than `CoChangeRatioWithoutDep(f₁,f₂)` in most cases. This intuition was formalized as an experiment with the following hypotheses:

Null hypothesis (H_0): Occurrence of co-changes involving f_1 and f_2 is not higher when f_1 depends on f_2.

Alternative Hypothesis (H_1): Occurrence of co-changes involving f_1 and f_2 is higher when f_1 depends on f_2.

Our findings for this research question are presented in Section IV.B.

2) *RQ2 – Data Processing and Analysis*

To answer this research question, it becomes imperative to be able to identify the different kinds of changes. In the scope of this paper, we focus on investigating the extent to which the three following kinds of change propagate:

(i) *Added methods* and *changed methods*: in a commit, a developer might add methods to a class and change existing methods. For instance, the method `foo()` from the class A might be modified to call the method `bar()` from class B. To the purpose of this analysis, constructors and the set of lines corresponding to the declaration of attributes are both considered methods.

(ii) *Added types*: in a commit, a developer might add a new type to either a new class or an existing class.

(iii) *Types with changed set of type-to-type (t2t) dependencies*: in a commit, a developer might add or remove a type dependency for a certain type s/he is working on. For instance, a developer might make a certain class A extend a class B or make a class C implement an interface D instead of an interface E.

The following algorithm describes the approach we employed to detect added and changed methods (i).

Algorithm II: Capturing added and changed methods

```
for each commit c do
  code <- checkout(c)
  prevCode <- checkout(previous(c))
  for each file f in commit c do
    methods <- getMethods(f,code)
    prevMethods <- getMethods(f,prevCode)
    methodComp <- MethodComparator.run(
       methods,prevMethods)
    record(c,methodComp)
```

Subroutine:
`MethodComparator.run(methods,prevMethods)`

```
identical <- removeIdentical(
   methods,prevMethods)
changed <- removeChanged(
          methods,prevMethods)
03.   deleted <- prevMethods
04.   added <- methods
05.   methodComp <- new MethodComp(
          identical,changed,deleted,added)
06.   return methodComp
```

We start by getting the list of methods from a certain artifact in the current code snapshot and in the previous code snapshot (line 01-06). We then compare the two lists of methods (line 07) and record the changes (line 08). We detect added and removed methods using a simple syntactic heuristic. First, identical methods are removed from both lists (subroutine, line 01). We then capture changed methods (subroutine, line 02). We consider that a method m changed into a method m' when their signatures are the same, their return type is the same and their body is different. We then remove these changed methods from both lists. The remaining methods in the `prevMethods` list are deemed as deleted (subroutine, line 03). The remaining ones in the `methods` list are deemed as added.

With relation to finding the added types (ii), we proceed in a similar fashion of Algorithm II. Let the types found in the previous version of a file be `prevTypes` and let the types found in the current version of a file be `types`. We deem two types as identical when they have the exact same source code. Let `identicalTypes` be the set with the identical types. The set of added types correspond to `types` minus `identicalTypes`.

Finally, in order to detect changes in type-to-type dependencies (iii), we employed Algorithm III.

Algorithm III: Capturing types with changed set of type-to-type (t2t) dependencies

```
01. for each commit c do
02.   code <- checkout(c)
03.   prevCode <- checkout(previous(c))
04.   for each file f in commit c do
05.     changedTypes <- empty list
06.     types <- getTypes(f,code)
07.     prevTypes <- getTypes(f,prevCode)
08.     pairs<prevType,type) <- types with same
          FQN, but different source code
09.     for each pair in pairs do
10.       if (pair.prevType.getT2TDeps ≠
            (pair.type.getT2TDeps)
11.         changedTypes.add(type)
12.     record(c,changedTypes)
```

We first get the list of types from a certain artifact in the current code snapshot and in the previous code snapshot (line 01-07). We then discover the types with same FQN but with different source code (line 08). Next, we examine these (pairs of) types to discover the cases where type-to-type dependencies were added or removed (line 09-11). Finally, we record the results (line 12).

Now that we showed how the three kinds of changes we defined were detected, we can rely on Algorithm I to answer the research question. However, the way the client is determined changes now (line 07). Instead of checking whether there is any dependency from f_1 to f_2, we check the low-level software changes (i), (ii), and (iii) we previously defined. More specifically, we perform the steps described in Algorithm IV.

Algorithm IV: Deciding whether there is a dependency from f_1 to f_2 while considering the change context

```
01.  deps <- JDX.getAllDeps($f_1$,$f_2$)
02.  for each dep do
03.    addedM <- get added methods in commit
04.    removedM <- get removed methods in commit
05.    if (addedM or changedM contains dep.Supp) do
06.      isClient <- true
07.      break
08.    addedT <- get added types in commit
09.    changedT <- get types with changed t2t deps
10.    if (addedT or changedT contains dep.Supp) do
11.      isClient <- true
12.      break
13.  return isClient
```

The main idea is to discover whether f_1 has a dependency to something that changed in f_2. Hence, we first get all kinds of dependencies from f_1 to f_2 (line 01). Next, we check whether the supplier of any of these dependencies is either an added or changed method (lines 03-07) or an added or changed type (lines 08-12). If we find at least a single positive case, then we state that f_1 is a client of f_2.

Finally, just as we did for RQ1, we check whether the rate with which f_1 co-changes with f_2 is higher when f_1 structurally depends on f_2 (as compared to when f_1 does not depend on f_2). This intuition is again formalized as an experiment with the following hypotheses:

Null hypothesis (H_0): Occurrence of co-changes involving f_1 and f_2 is not higher when f_1 depends on f_2 and the change context is taken into account.

Alternative Hypothesis (H_1): Occurrence of co-changes involving f_1 and f_2 is higher when f_1 depends on f_2 and the change context is taken into account.

Our findings for this research question are presented in Section IV.C.

3) *RQ3 – Data Processing and Analysis*

To answer this research question, we identified the changes that happened in a commit and inferred whether they propagated via structural dependencies. For the purpose of this analysis, we focused on added and changed methods. The detailed steps we followed are described in Algorithm V.

Algorithm V: Determining change propagation via structural dependencies

```
01. for each commit c do
02.   code <- checkout(c)
03.   addedChangedM <- discover added and
        changed methods
04.   if (addedChangedM > 0) do
05.     deps <- JDX.getCallDeps(code)
06.     callGraph <- Jung.getGraph(deps)
07.     callGraphT <- Jung.getTransitiveClosure(
          callGraph)
08.     propagatedM <- 0
09.     for each method m in addedChangedM do
10.       preds <- callGraphT.getPred(m)
11.       for each method pred in preds do
12.         if(m ≠ pred and
13.           addedChangedM constains pred) do
14.           distance <- Dijkstra.calcDistance(
                callGraph,pred,m)
```

```
15.           if (distance <= 3) do
16.             propagatedM <- propagated + 1
17.             break
18.       percentProp <- propagated / size(
            addedChangedM)
19.       record(c,percentProp)
```

For each commit, we check-out the code (line 02) and discover the added and changed methods (line 03) using the approach we described in Section IV.C. If we find at least one added or changed method, we proceed with the analysis. Otherwise, we skip the commit and investigate the following one (line 04). Using JDX, we obtain all call dependencies found in the code (line 05). With these dependencies in hands, we use Jung to build a *call-graph* (line 06). A call-graph is a directed graph where vertices are methods and edges represent calling relationships. More specifically, the edges connect a method m to another method m' if and only if there is at least one call dependency from m to m'. Subsequently, we use Jung once more, but this time to calculate the *transitive closure* of the call-graph (line 07). The transitive closure is a graph in which there is an edge connecting v to v' if and only if there is a path from v to v' in the original graph. In our case, the transitive closure of the call-graph is a directed graph in which an edge from m to m' implies in the existence of a transitive dependency in the original call-graph. In line 08, we initialize a counting variable that will record the number of added/changed methods that propagated changes via call dependencies. We consider that an added/changed method m propagates a change via call dependencies if at least one of its predecessors in the transitive closure is a method m' that was either added or changed in the commit. However, since an edge in the transitive closure stands for a transitive dependency, we check back in the original call-graph how many edges (hops) were needed to connect m' to m. We thus use the Dijkstra's shortest path algorithm (line 14) to determine the length of the shortest path that connects m' to m in the original call-graph. However, since this path might be quite lengthy, we restrict the maximum distance to 3 (line 15). In other words, we accept only two intermediary methods in the path that connects m' to m. The rationale is the following: if too many hops are needed to connect m' to m, then it is more likely that they changed together because of some other reason that is not directly related to the calling dependencies. For instance, it could be the case that two classes are semantically related, with similar terms present in their comments and identifiers (a.k.a., conceptual coupling [17]). Finally, we record the proportion of added/changed methods that propagated changes in the commit (line 19). Once all data are captured, we analyze the distribution using descriptive statistics. Our findings for this research question are presented in Section IV.D.

F. Supporting Tools

XFlow. This is an extensible, interactive, and stand-alone tool we have developed [37], whose general goal is to provide a comprehensive software evolution analysis by mining software repositories and taking into account both technical and social aspects of the developed systems. In this study, we employed XFlow to parse and store commit logs. Project website: https://github.com/golivax/xflow2.

JDX. Java Dependency eXtractor (JDX) is a core tool in this study. It is a Java library we have developed to extract dependencies from Java code. The tool is robust and is able to extract all dependencies listed in Section II.B. The library relies on the robust Java Development Tools Core (JDT Core) library, which is the incremental compiler used by the Eclipse IDE. As a desirable consequence, JDX is able to handle Java source code in its plain form. Project website: https://github.com/golivax/JDX

Jung. Java Universal Network/Graph Framework (Jung) is a Java library that provides a common and extensible language for modeling, analyzing, and visualizing data that can be represented as a graph or network. In this study, we employed Jung to help us compute certain algorithms, such as the transitive closure of a directed (dependency) graph. Project website: http://jung.sourceforge.net

Structural Dependencies Analyzer. This is a custom tool we built for the purpose of answering the research questions of this study. It relies on JDX and Jung. It implements all algorithms shown in Section III.E.

R and RStudio. All statistical analyses of this study were conducted using the R package v3.1.2., with the support of RStudio IDE v.0.98.1103.

IV. Study Results

This section presents and discusses the results of our three research questions. For each research question, before presenting the results, we first briefly recall our motivation to study it and the approach we employed to answer it. All data and scripts used in this study are available at: https://github.com/golivax/issre2015.

A. Preliminary Analysis: Characterizing Subject Systems

Given the criteria listed in Section III.B, we selected the 4 following projects: Apache Lucene (core), Apache Tomcat 7, Megamek, and Apache Commons CSV. Lucene is an information retrieval library widely used in the implementation of Internet search engines. Tomcat is a renowned web server and servlet container that implements several Java EE specifications, Megamek is a turn-based strategy game inspired by the "Classic BattleTech" board game. Commons CSV is a library to read, write, and manipulate CSV files.

Table II shows the commit interval we mined for each system. It also depicts the repository URL, the repository path regex, and the total number of commits that matched the regex. We say that a commit matches the regex when it contains at least one file whose path matches the regex. This implies that commits with no Java files were discarded. We also highlight that the reason why the commit intervals seem so big for the Apache projects is because all projects from the Apache Software Foundation are hosted in a single Subversion server. Therefore, we must take a large interval to find all the commits we want for a single system.

B. RQ1: Are dependent files more likely subject to co-change than independent ones?

Motivation. Classic Software Engineering literature has long stated that structural coupling should be minimized because every time a certain class changes, all other dependent classes are also likely to change, thus inducing ripple effects [2, 3, 8, 10]. Quoting Stevens et al. [20]:

TABLE II. SUBJECT SYSTEMS

Project	Commit Interval	VCS URL	Mined Path (regex)	Total Commits (w/ Java files)
Apache Lucene (core)	[149570, 926576]	https://svn.apache.org/repos/asf/	/lucene/java/trunk/src/java/.*?\\.java	1962
Apache Tomcat 7	[540106, 1155255]	https://svn.apache.org/repos/asf/	/tomcat/trunk/java/.*?\\.java	3554
Megamek	[2, 11344]	http://svn.code.sf.net/p/megamek/code	/trunk/megamek/src/megamek/.*?\\.java	7754
Apache Commons CSV	[1299104, 1603967]	https://svn.apache.org/repos/asf/	/commons/proper/csv/trunk/src/main/java/.*?\\.java	415

"Minimizing connection between modules also minimizes the paths along which changes and errors can propagate into other parts of the system, thus eliminating disastrous ripple effects, where changes in one part cause errors in another, necessitating additional changes elsewhere, giving rise to new errors, etc."

Hence, with this research question, we aim to empirically investigate whether dependent files are more likely subject to co-change than independent ones.

Approach. To address this research question, we mine the version control system. For each commit, we search for co-changes and structural dependencies between pairs of files <f_1,f_2>, where f_2 is a file in the commit's change-set and f_1 is a file in the checked-out code. Algorithm I describes the steps we followed in details.

Results: We start by showing the results of the comparison of CD and $\overline{\text{C}}$D. For this analysis, we selected only the file pairs where `Dep(f1, f2)` $\neq 0$, i.e., file pairs where f_1 was a client of f_2 in at least one commit. The results for the subject systems are summarized as bean plots in Figure 3. The black horizontal lines denote the median of the distributions. The dotted line represents the median of the two distributions combined.

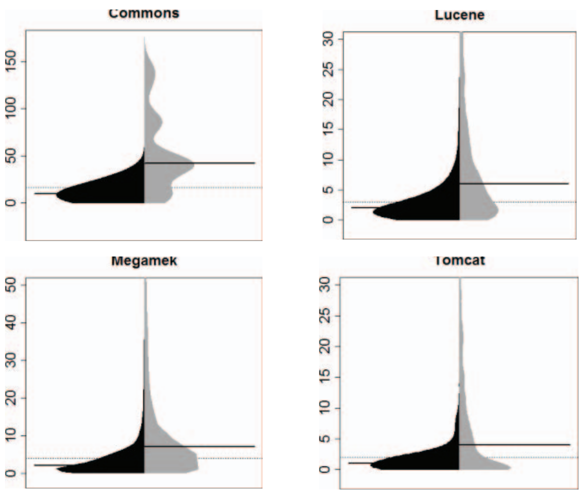


Fig. 3. Beanplots comparing the distribution of CD (black) and $\overline{\text{C}}$D (gray).

In all systems, the median of CD (left, black) is smaller than the median of $\overline{\text{C}}$D (right, gray). Moreover, the distribution of $\overline{\text{C}}$D has a much larger tail. This suggests that CD is in general smaller than $\overline{\text{C}}$D. To further investigate this, we performed a paired two-sample Wilcoxon test (a.k.a. Mann-Whitney test) with the alternative hypothesis that CD is less than $\overline{\text{C}}$D at a significance level of 0.05. We could reject the null hypothesis in all systems with the following p-values: less than 2.2e-16 (Lucene), less than 2.2e-16 (Megamek), equal to 6.506e-06 (CSV Commons), and less than 2.2e-16 (Tomcat).

In summary, we found a consistent behavior across all systems: given a pair of files <f_1,f_2> where f_1 depends on f_2, the number of co-changes (CD) is, in general, *smaller* than the number of no co-changes ($\overline{\text{C}}$D).

(R.I) The number of times f_1 co-changes with f_2 is *smaller* than the number of times f_1 does not co-change with f_2 in the context where f_1 depends on f_2 and f_2 changes.

Next, we investigate whether dependent files are more likely subject to co-change than independent ones. The `CoChangeRatioWithDep(f1, f2)` metric is undefined when both CD and $\overline{\text{C}}$D are zero. Analogously, `CoChangeRatioWithoutDep(f1, f2)` is undefined when both C$\overline{\text{D}}$ and $\overline{\text{CD}}$ are zero. Therefore, for each project, we had to first preprocess the data to remove the cases where these metrics could not be calculated. This led to two samples s_1 and s_2:

- Sample s_1: `CoChangeRatioWithDep(f1, f2)` with $\text{Dep}(f_1, f_2) = \text{CD} + \overline{\text{C}}\text{D} > 0$
- Sample s_2: `CoChangeRatioWithoutDep(f1, f2)` with $\text{NoDep}(f_1, f_2) = \text{C}\overline{\text{D}} + \overline{\text{CD}} > 0$

As we did in the previous case, we start with a graphical analysis of the distributions of s_1 and s_2 for each project. Figure 4 depicts the bean plots we obtained.

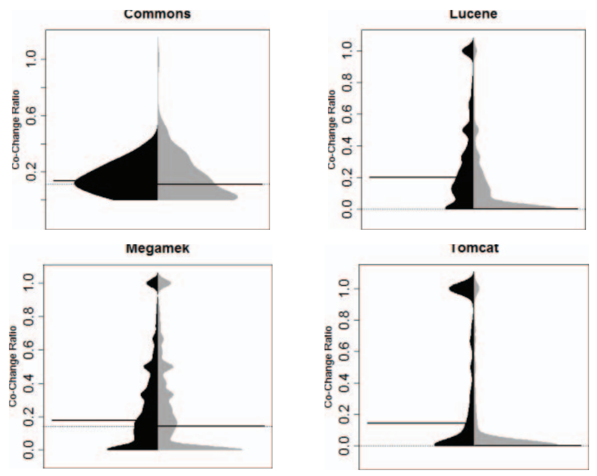


Fig. 4. Beanplots comparing the distribution of s_1 (black) and s_2 (gray).

In all systems, the median of s_1 is higher than the median of s_2. This supports our hypothesis that the rate with which f_1 co-changes with f_2 is higher when f_1 structurally depends on f_2. However, we highlight that the medians for s_1 are all fairly low:

13.54% for Commons CSV, 20% for Lucene, 17.65% for Megamek, and 14.29% for Tomcat.

The next step was to evaluate whether dependent files are more likely subject to co-change than independent ones (alternative hypothesis). We employed a non-paired two-sample Wilcoxon test at a significance level of 0.05. As opposed to the first test we conducted, this must be *non-paired* because of the preprocessing we executed. We could reject the null hypothesis for three systems, all of them with a p-value smaller than 2.2e-16. The exception was Apache Commons CSV, since its p-value was equal to 0.1811. In fact, in this system, the medians of s_1 and s_2 were very similar: 0.1354 for s_1 and 0.1129 for s_2. Hence, in this system, structural dependencies seem not to be associated with co-changes. Given that Commons CSV is a much smaller system (10 classes in v1.2) and it is a library, our hypothesis is that it has very stable interfaces, leading to surgical changes that are not driven by structural relationships.

In summary, in three of the four systems, we could reject the null hypothesis. This provides empirical evidence to support the claim that dependent files are more likely subject to co-change than independent ones.

(R.II) The rate with which f_1 co-changes with f_2 is higher when f_1 structurally depends on f_2 (as compared to when f_1 does not depend on f_2). However, this rate is fairly low, ranging from 13.5% to 20%.

As a general conclusion, we can say that even though it is more likely that f_1 and f_2 will not co-change just because f_1 depends on f_2 (i.e., dependencies do not instantly make two files change together), the rate with which f_1 co-changes with f_2 is higher when f_1 structurally depends on f_2.

C. *RQ2: Are dependent files more likely subject to co-change than independent ones when the change context is taken into account?*

Motivation. The previous analysis did not take into consideration the specific changes that happened in each file of the commit's change-set. That is, we analyzed dependencies at the file-level. What if we accounted for the specific changes? If a certain method is changed, will its clients (from other classes) change as well? If a certain type now extends another, will the clients of the former be affected?

Approach. The approach we employed is similar to that of RQ1. However, the way the structural client is determined is different. Instead of checking whether there is any dependency from f_1 to f_2 in a given commit, we check whether there is a dependency from f_1 to "something" that changed inside f_2. In the scope of this paper, "something" can be either: (i) added/changed methods (e.g., a new method `m` was added to a type `t` in f_2), (ii) added types (e.g., a new type `t` was added to f_2), and type-to-type dependencies (e.g., a type `t` in f_2 now extends type `t'` instead of `t''`).

Results. We proceed in the exact same way we did for RQ1. Therefore, we start by showing the results of the comparison of CD and $\overline{\text{CD}}$ via bean plots (Figure 5). We highlight that we again selected only the file pairs where $\text{Dep}(f_1, f_2) \neq 0$, i.e., file pairs where f_1 was a client of f_2 in at least one commit.

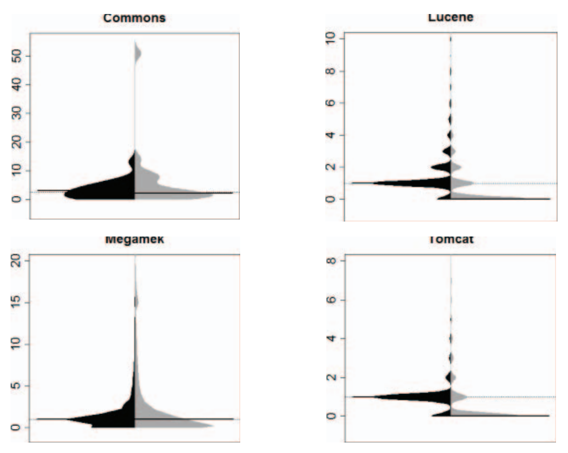


Fig. 5. Beanplots comparing the distribution of CD (black) and $\overline{\text{CD}}$ (gray).

As compared to RQ1, the results are much subtler now, as the medians became much closer to each other in all systems. This is also evidenced by the descriptive statistics for the distributions, which are shown in Table III. Lucene and Tomcat seem to show a similar behavior, with higher number of observations equaling to zero in the right-hand side ($\overline{\text{CD}}$) and a higher number of observations equaling to one in the left-hand side (CD). In both systems, the median for CD is one, the median for $\overline{\text{CD}}$ is zero, and the mean of CD is higher than the mean of $\overline{\text{CD}}$. The shape of the distributions of Commons CSV and Megamek are quite different from the two other systems. In Commons CSV, even though the median of CD is higher than that of $\overline{\text{CD}}$, the mean is not. In Megamek, the medians of CD and $\overline{\text{CD}}$ are identical, and the mean of $\overline{\text{CD}}$ is higher.

TABLE III. DESCRIPTIVE STATISTICS FOR CD AND $\overline{\text{CD}}$

Systems	CD						$\overline{\text{CD}}$					
	Min	*Q1*	*Med*	*Mean*	*Q3*	*Max*	*Min*	*Q1*	*Med*	*Mean*	*Q3*	*Max*
Commons CSV	0	1	3	3.04	4	13	0	1	2	5.81	6.75	51
Megamek	0	0	1	1.76	2	305	0	0	1	5.12	3	504
Lucene	0	1	1	1.55	2	31	0	0	0	0.98	1	45
Tomcat	0	1	1	0.97	1	16	0	0	0	0.71	1	38

To further investigate the data, we performed a series of paired two-sample Wilcoxon tests, all at a significance level of 0.05. For Lucene and Tomcat, we performed the test with the alternative hypothesis that CD is *greater than* $\overline{\text{CD}}$. In both cases, we could reject the null hypothesis with a p-value small than 2.2e-16. For Commons CSV, we tested the alternative hypothesis that CD is *less than* $\overline{\text{CD}}$. The p-value was 0.1562 and thus we could not reject the null hypothesis. In addition, we tested the alternative hypothesis that CD and $\overline{\text{CD}}$ are distributions with different shapes (two-sided test). The p-value was 0.3124 and thus we could not reject the null hypothesis again. That is, for Commons CSV, the distributions of CD and $\overline{\text{CD}}$ are roughly indistinguishable (as also supported by its bean plot depicted in Figure 5). Finally, in Megamek, since the median of CD and $\overline{\text{CD}}$ are the same, we first tested the alternative hypothesis that CD and $\overline{\text{CD}}$ are distributions with different shapes. We could reject the null hypothesis with a p-value smaller than 2.2e-16., i.e., the location shift is not equal to zero. We then tested whether CD was less than $\overline{\text{CD}}$, since the mean of CD (1.76) is smaller than the mean of $\overline{CD}$ (5.12). We could again reject the null hypothesis with the same p-value.

In summary, we compared CD and $\overline{CD}$ to understand the frequency of co-changes in the occurrence of structural dependencies. Given a pair of files <f_1,f_2> with f_1 depending on f_2, the number of co-changes is, in general, *greater* than the number of no co-changes (absence of co-changes) for Lucene and Tomcat. In Commons CSV, these numbers are roughly the same. In Megamek, the number of co-changes is, in general, *smaller* than the number of no co-changes. Hence, the results changed substantially as compared to the previous RQ.

(R.III) When taking the change context into account, the results change substantially as compared to (R.I). In the context where f_1 depends on f_2 and f_2 changes: (a) It is more likely that f_1 *will* co-change with f_2 (instead of not co-changing) in Lucene and Tomcat; (b) the number of co-changes and no co-changes will be roughly the same in Commons CSV; (c) it is more likely that f_1 *will not* co-change with f_2 in Megamek

We now investigate whether dependent files are more likely subject to co-change than independent ones. As we did for RQ1, we check whether the rate with which f_1 co-changes with f_2 is higher when f_1 structurally depends on f_2 (as compared to when f_1 does not depend on f_2). We apply the same preprocessing and obtain the samples s_1 and s_2, which are the target of the analysis. Before evaluating the hypothesis, we start with a graphical analysis of the distributions of s_1 and s_2 for each project. Figure 6 depicts the bean plots we obtained.

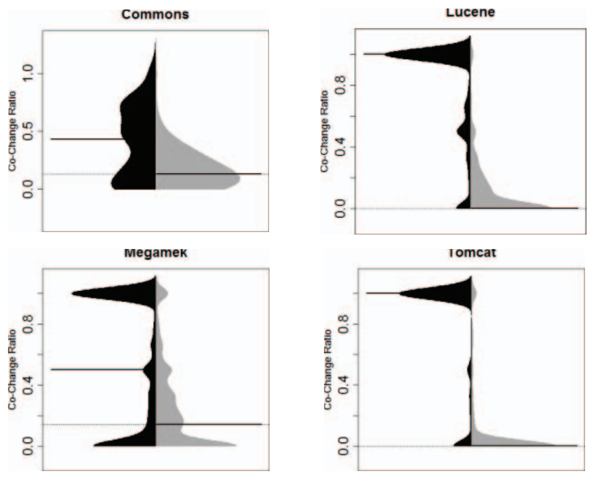


Fig. 6. Beanplots comparing the distribution of s_1 (black) and s_2 (gray).

In all systems, the median of s_1 is significantly higher than the median of s_2. In particular, the median of s_1 is equal to 1.0 in Lucene and Tomcat. This implies that at least 50% of the observations in these distributions are equal to 1, i.e., the client always changed when the supplier changed (the change always propagated). In Megamek, the third quartile of s_2 is equal to 1.0. In turn, the shape of the distributions for Commons CSV looks more uniform, with data more evenly spread in the [0,1] interval. We highlight that the medians for s_1 are significantly higher than those we found in RQ1, as they now ranged from 43.08% (Commons CSV) to 100% (Lucene and Tomcat).

We then evaluated the hypothesis itself using a non-paired two-sample Wilcoxon test at a significance level of 0.05. We could reject the null hypothesis for all systems. Lucene, Megamek, Tomcat had a p-value smaller than 2.2e-16. Commons CSV had a p-value of 0.0015, which is not as strong as the former, but still sufficient to reject the null hypothesis.

In summary, all these results provide empirical evidence to support the claim that dependent files are more likely subject to co-change than independent ones.

(R.IV) When taking the change context into consideration, the rate with which f_1 co-changes with f_2 is higher when f_1 structurally depends on f_2 (as compared to when f_1 does not depend on f_2). This rate is also substantially higher than that we found when ignoring the change context (R.II).

D. RQ3: Do changes propagate via structural dependencies?

Motivation. Even though structural dependencies might lead to co-changes, it is not clear the extent to which the architecture "justify" software changes [22]. How do the actual software changes relate to the architecture? In particular, are there situations where two classes change together and there is no structural connection between them? How often does it happen? These are the aspects we will investigate.

Approach. To answer this research question, we identified the changes that happened in a commit and inferred whether they propagated via structural dependencies using graph algorithms. For the purpose of this analysis, we focused on added and changed methods. The detailed steps we followed are described in Algorithm V.

Results. Figure 7 depicts the bean plot for the proportion of added/changed methods that propagated changes via call dependencies per commit. We see a consistent behavior across all systems, as their distributions are very similar. Most importantly, the distributions are very right-skewed and the medians are all zero.

Table IV depicts descriptive statistics for the same variable. The means vary from 11.40% (Tomcat) to 18.55% (Lucene), and the standard deviations are significant, varying from 19.57% to 22.77%. Hence, the mean is also fairly low for all distributions and there is a significant variability around it.

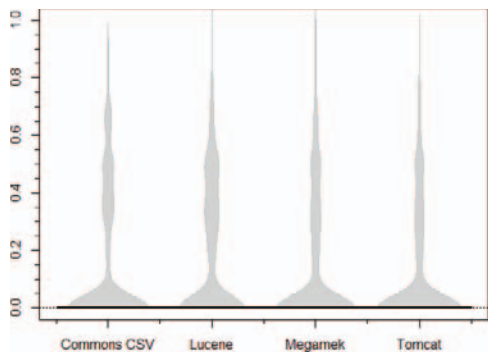


Fig. 7. Bean plot for the proportion of methods that propagated changes per commit.

TABLE IV. DESCRIPTIVE STATISTICS FOR THE PROPORTION OF METHODS THAT PROPAGATED CHANGES PER COMMIT

Systems	Min	Q1	Med	Mean	StdDev	Q3	Max
Commons CSV	0	0	0	14.46%	22.77%	33.33%	86.36%
Megamek	0	0	0	13.72%	21.19%	27.27%	100.00%
Lucene	0	0	0	18.55%	22.71%	36.27%	100.00%
Tomcat	0	0	0	11.40%	19.57%	20.00%	88.89%

These results have strong implications in practice: most changes in methods do not tend to propagate via call dependencies. This supports the claim found in the study of Geipel and Schweitzer [6]: very few changes are "justified" by the software architecture.

(R.V) Changes in methods rarely propagate via call dependencies.

V. RELATED WORK

Several recent studies regarded the dependency structure as an indicator, predictor, or metric for co-changes in software systems [1, 7, 18]. Sangal et al. [19] even call certain modules "change propagators", which are the ones that depend on many other modules and are also used by many other modules. However, the interplay between structural dependencies and co-changes was little investigated from an empirical perspective.

To the best of our knowledge, the work of Geipel and Schweitzer [6] is the most similar to ours, as they also investigate the link between structural dependencies and co-changes (change propagation) in Java systems. Although they investigated many more systems (35), they only considered dependencies at the file-level and disregarded the change context (as opposed to what we did in RQ2). Most importantly, they only take into consideration the latest code snapshot when extracting dependencies, arguing that 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`. We found that this assumption did not hold for the projects we investigated. Hence, as opposed to Geipel and Schweitzer, we recalculate dependencies every time we check-out a new version of the code. Finally, all the systems they investigated were hosted in CVS, while ours were all hosted in SVN. Differently from CVS, SVN supports atomic commits and co-changes can thus be inferred from the change-sets. In CVS, change-sets need to be inferred using heuristics, like the sliding time window [23]. Hence, we believe our data set is more reliable.

Fluri et al. [5] investigated the degree to which co-changes are caused by structural changes (which they also call *source code coupling*) and textual modifications (e.g., software license updates and whitespaces between methods spaces). A preliminary evaluation involving the *compare plugin* of Eclipse showed that more than 30% of all change transactions did not include any structural change. Therefore, more than 30% of all change transactions have nothing to do with structural coupling. They also found that more than 50% of change transactions had at least one non-structural change. They hypothesize that this could be the result of code ownership/commit habit (a developer works all day in his files and commits everything by the end of the day) and frequent license changes.

In a broader context, Hassan and Holt [7] studied how change propagation can be predicted. They investigated how a change in one source code entity (function, variable, etc.) propagates to other entities. As opposed to our study, they investigated C systems and only three kinds of relationships: *call*, *use*, and *define*. Later on, Malik and Hassan produced a new version of the study [9], in which they described a hybrid and adaptive change propagation predictor that relies on both the software structure and in the development history.

VI. THREATS TO VALIDITY

Some aspects may have pose threats to the validity of this study. First, our toolset has some limitations. Algorithm I relies on paths for distinguishing between files. Hence, it does not account for renames and file moving. Furthermore, in RQ2 we do not deal with removed methods. Although method removals could be the trigger of change propagations chains, we believe that adding and changing methods are more usual operations done by developers. Hence, we believe our results are still reliable, in the sense that they cover enough ground.

In RQ2, we deem "changed types" as those that had their type-to-type dependencies changed. As part of our future work, we plan to use more sophisticated heuristics. In RQ3, we only investigated calling relationships. Analogously, in future work, we plan to investigate other kinds of structural relationships. Moreover, even though we selected projects of different sizes and domains, the small number of subjects limits the generalizability of our results. Furthermore, we have only investigated a single module of a single branch of each project. Finally, given the research method we used, we cannot infer causal relationships between dependencies and co-changes.

VII. CONCLUSIONS

Software development is an inherently complex socio-technical activity. Since the notorious software crisis acknowledged in the late 60's [11], practitioners and researchers have sought better ways to develop software systems. During this quest, some fundamental Software Engineering principles have emerged. One of these principles is known as "low coupling" and says that both the number and the strength of interconnections among modules should be minimized in order to prevent change propagation.

In this study, we set out to empirically investigate the link between dependencies and change propagation. With the support of a fairly sophisticated toolset, we analyzed 4 open source systems written in Java. Our results indicated that, in general, it is more likely that two 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 we track down dependencies to the low-level entities that are changed in commits. We also found several cases where software changes could not be justified using structural dependencies, meaning that co-changes are possibly induced by other subtler kinds of relationships

Besides addressing some of the limitations given in the prior section, as future work we plan and factorize the analysis done in RQ2 according to the different kinds of structural dependencies, including *method calls*, *imports*, and others. This would allow us to know the role of each kind of dependency separately.

ACKNOWLEDGMENT

Gustavo Oliva and Marco Gerosa receive individual grants from CNPq ("Conselho Nacional de Desenvolvimento Científico e Tecnológico"). The authors would like to thank Professor Ahmed E. Hassan for his feedback in very early versions of this study.

REFERENCES

[1] Abdeen, H., Bali, K., Sahraoui, H. and Dufour, B. 2015. Learning Dependency-based Change Impact Predictors using Independent Change Histories. *Information and Software Technology*. (2015).

[2] Arnold, R.S. 1996. *Software Change Impact Analysis*. IEEE Computer Society Press.

[3] Booch, G., Maksimchuk, R.A., Engel, M.W., Young, B.J., Conallen, J. and Houston, K.A. 2007. *Object-Oriented Analysis and Design with Applications*. Addison-Wesley Professional.

[4] Brooks, F.P. 1995. *The Mythical Man-Month: Essays on Software Engineering*. Addison-Wesley Professional.

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

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

[7] Hassan, A.E. and Holt, R.C. 2004. Predicting Change Propagation in Software Systems. *Proceedings of the 20th IEEE International Conference on Software Maintenance* (Washington, DC, USA, 2004), 284–293.

[8] Larman, C. 2004. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development*. Prentice Hall.

[9] Malik, H. and Hassan, A.E. 2008. Supporting software evolution using adaptive change propagation heuristics. *IEEE International Conference on Software Maintenance, 2008. ICSM 2008.* (2008), 177–186.

[10] Martin, R.C. and Martin, M. 2006. *Agile Principles, Patterns, and Practices in C#*. Prentice Hall.

[11] Naur, P. and Randell, B. eds. 1969. *Software Engineering: Report of a conference sponsored by the NATO Science Committee, Garmisch, Germany, 7-11 Oct. 1968*. NATO Science Committee.

[12] Oliva, G.A. and Gerosa, M.A. 2015. Experience Report: How do Structural Dependencies Influence Change Propagation? An Empirical Study. *Proceedings of the 26th IEEE International Symposium on Software Reliability Engineering* (Gaithersburg, USA, 2015).

[13] Oliva, G.A. and Gerosa, M.A. 2011. On the Interplay between Structural and Logical Dependencies in Open-Source Software. *Proceedings of the 2011 25th Brazilian Symposium on Software Engineering* (Washington, DC, USA, 2011), 144–153.

[14] Oliva, G.A., Santana, F.W.S., Gerosa, M.A. and Souza, C.R.B. de 2012. Preprocessing Change-Sets to Improve Logical Dependencies Identification. *Proceedings of the 6th International Workshop on Software Quality and Maintainability* (Szeged, Hungary, 2012).

[15] Page-Jones, M. 1992. Comparing Techniques by Means of Encapsulation and Connascence. *Commun. ACM*. 35, 9 (Sep. 1992), 147–151.

[16] Parnas, D.L. 1994. Software aging. *Proceedings of the 16th international conference on Software engineering* (Sorrento, Italy, 1994), 279–287.

[17] Poshyvanyk, D. and Marcus, A. 2006. The Conceptual Coupling Metrics for Object-Oriented Systems. *ICSM* (2006), 469–478.

[18] Rajlich, V. 1997. A Model for Change Propagation Based on Graph Rewriting. *Proceedings: 1997 International Conference on Software Maintenance (ICSM 1997)* (1997), 84–91.

[19] Sangal, N., Jordan, E., Sinha, V. and Jackson, D. 2005. Using Dependency Models to Manage Complex Software Architecture. *Proceedings of OOPSLA'05* (2005), 167–176.

[20] Stevens, W.P., Myers, G.J. and Constantine, L.L. 1974. Structured design. *IBM Syst. J.* 13, 2 (Jun. 1974), 115–139.

[21] Wirth, N. 1971. Program Development by Stepwise Refinement. *Commun. ACM*. 14, 4 (Apr. 1971), 221–227.

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

[23] Zimmermann, T. and Weißgerber, P. 2004. Preprocessing CVS Data for Fine-Grained Analysis. *Proceedings 1st International Workshop on Mining Software Repositories (MSR 2004)* (Los Alamitos CA, 2004), 2–6.