Towards a Classification of Logical Dependencies Origins: A Case Study

Gustavo A. Oliva
Comp. Science Dept./IME
University of São Paulo
CEP: 05508-090
São Paulo – SP – Brazil
goliva@ime.usp.br

Francisco W.S. Santana
Computing Department
Federal University of Pará
CEP: 66075-100
Belém – PA – Brazil
wertherjr@gmail.com

Marco Aurélio Gerosa
Comp. Science Dept./IME
University of São Paulo
CEP: 05508-090
São Paulo – SP – Brazil
gerosa@ime.usp.br

Cleidson R.B. de Souza
IBM Research
CEP: 04007-05
São Paulo – SP – Brazil
cleidson.desouza@acm.org

ABSTRACT

Logical dependencies are implicit relationships established between software artifacts that have evolved together. Software engineering researchers have investigated this kind of dependency to assess fault-proneness, detect design issues, infer code decay, and predict likely changes in code. Despite the acknowledged relation between logical dependencies and software quality, the nature of the logical dependencies is unknown in the literature. Most authors hypothesize about their origins, but no empirical study has been conducted to investigate the real nature of these dependencies. In this paper, we investigated the origins of logical dependencies by means of a case study involving a Java FLOSS project. We mined the project repository, filtered out irrelevant data based on statistical analyses, and performed a manual inspection of the logical dependencies to identify their origins using information from the revision comments, code diffs, and informal interviews held with the developers of the analyzed project. Preliminary results showed that logical dependencies involved files that changed together for a series of different reasons, which ranged from changing software license to refactoring classes that belonged to the same semantic class.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement – *reestructuring, reverse engineering, and reengineering; version control*.

General Terms

Measurement, Design, Experimentation.

Keywords

Software evolution, mining software repositories, logical dependencies, logical coupling, change coupling, empirical software engineering, case study.

1. INTRODUCTION

Software Engineering researchers have long recognized the relationship between code structural dependencies, or static dependencies, and software failures [1, 2, 3, 4]. Recently, research on software evolution has introduced a novel approach for dependency identification that reveals new and more subtle relationships between software artifacts. The concept underlying this notion is known as logical dependencies [5, 6], which refers to evolutionary dependencies that are established among source-code files that are frequently changed together (although not necessarily structurally related). Graves *et al*. [7] showed that past changes are good predictors of future faults. Mockus and Weiss [8] found out that the spread of a change over subsystems and files is a strong indicator that the change will contain a defect. Cataldo and colleagues [9] reported through a detailed empirical study that the effect of logical dependencies on fault proneness was complementary and significantly more relevant than the impact of structural dependencies in two software projects from different companies. Logical dependencies have also been employed to detect design issues [10], infer code decay [11], and predict changes in software artifacts [12].

While it seems clear that logical dependencies play a major role in software reliability and in other software properties (such as maintainability and evolvability), no empirical examinations have been undertaken to reveal the origins of these dependencies. According to Cataldo *et al.* [9], a more detailed understanding of the origins of logical dependencies is an important future research direction with implications in research areas such as software quality. In particular, among different studies with varying purposes, authors have formulated a series of informal hypotheses regarding these origins: cascading function calls [9], semantic dependencies [9], platform evolution [9], code duplication [13], and the use of dependency injection and reflection techniques [14].

The main goal of this paper is exactly to investigate the origins of logical dependencies. This is achieved through a case study [15, 16, 17] involving a project entitled Groupware Workbench[1] (GW) [18], which offers a component-based toolkit for the development of Web 2.0 collaborative systems. The project is written in Java and it has received contribution from twelve developers from different academic institutions. GW has been active since 2008 and it has been distributed as Free Software by IME-USP FLOSS Competence Center[2] since December of the same year.

We mined the GW Subversion (SVN) repository with the XFlow tool [19] and identified all existing logical dependencies between software artifacts. Subsequently, we filtered out data and manually investigated the origins of logical dependencies by reading commit comments, looking at code diffs and validating our results with GW developers. Preliminary results showed that the analyzed logical dependencies involved files that changed

IWPSE-EVOL'11, September 5–6, 2011, Szeged, Hungary.

[1] http://www.groupwareworkbench.org.br/

[2] http://ccsl.ime.usp.br/en

together for different reasons, which ranged from changing software license to refactoring classes that belonged to a same semantic class.

Our main **contributions** include (i) a systematic approach for logical dependencies identification, grouping, and classification that can be reused and adapted for future research, (ii) a classification of the origins of logical dependencies of a real system, (iii) the identification of research opportunities based on such classification, and (iv) the implementation of a modified version of the sliding time window algorithm in XFlow.

The rest of this paper is organized as follows. In Section 2, we introduce logical dependencies and related metrics. In Section 3, we describe the research method used as well as its planning. In Section 4, we present the study results including the classification of logical dependencies. In Section 5, we present the threats to the validity of this study. In Section 6, we present the related work. In Section 7, we state our conclusions and ideas for future work.

2. LOGICAL DEPENDENCIES

Logical dependencies (a.k.a., change dependencies, evolutionary dependencies, and co-changes) are implicit dependencies that happen between software artifacts that evolved together [5, 6]. These artifacts are not necessarily structurally related, since they are connected from an evolutionary point of view, i.e. they have often changed together in the past, so they are likely to change together in the future. Unlike structural dependencies analysis (a.k.a., static analysis), this technique spots dependencies between any kind of artifact that composes a system, including configuration files (such as XML and property files) and documentation. The identification of logical dependencies is usually performed by parsing and analyzing the logs of a version control system (VCS).

Logical dependencies are defined for pairs of files and are commonly treated as data mining association rules [12]. Formally, an association rule is an implication of the form $X_1 \Rightarrow X_2$, meaning that when X_1 occurs, X_2 also occurs. In this notation, X_1 and X_2 are two disjoint sets of items. Furthermore, X_1 and X_2 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 would indicate that if a customer buys A and B together, he or she is also likely to buy C. In the context of our study, a logical dependency from a file f_2 to another file f_1 is denoted by $F_1 \Rightarrow F_2$, i.e. an association rule in which the antecedent and consequent are both singleton sets containing f_1 and f_2 respectively.

Measures of interest and significance for association rules are usually given by support and confidence thresholds. In our study, the support measure denotes the number of times two artifacts were changed together. The confidence measure defines the degree to which artifacts are logically connected, thus characterizing the strength of the relation. These concepts have been formalized by Zimmerman *et al.* [12] and we have adapted them for atomic-commit featured VCSs as follows:

- **Frequency** of a set F in a set of commits C as $frq(C, F) = |\{c \mid c \in C, F \subseteq c\}|$.
- **Support** of a rule $X_1 \Rightarrow X_2$ by a set of commits C as $supp(C, F_1 \Rightarrow F_2) = frq(C, F_1 \cup F_2) = P(F_1 \cap F_2)$, i.e. the probability of finding both antecedent and consequent in the set of commits C.
- **Confidence** of a rule $F_1 \Rightarrow F_2$ as $conf(C, F_1 \Rightarrow F_2) = frq(C,F_1 \cup F_2) / frq(C,F_1) = P(F_2|F_1)$, i.e. the probability of finding the consequent of the rule in commits under the condition that these commits also contain the antecedent.

It should be noted that the confidence values for $F_1 \Rightarrow F_2$ and $F_2 \Rightarrow F_1$ are different. In the first case, the confidence value determines (by definition) the degree to which file f_2 is a client of file f_1. Analogously, in the second case, the confidence value determines the degree to which file f_1 is a client of file f_2. To illustrate this subtle difference, consider the example shown in Figure 1.

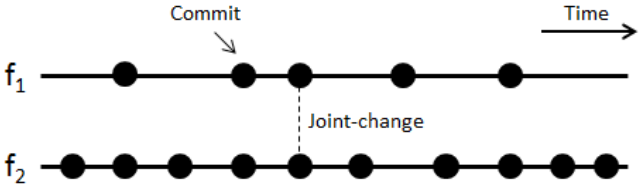

Figure 1. Association rule example

Most of the time, when f_1 is commited, f_2 is also commited. Therefore, the rule $F_1 \Rightarrow F_2$ (which states that f_2 depends on f_1) has a high confidence value of 4/5 = 0.8 = 80%. In contrast, the rule $F_2 \Rightarrow F_1$ (which states that f_1 depends on f_2) has a much lower confidence value of 4/10 = 0.4 = 40%.

3. RESEARCH METHOD

Case study is a well-established empirical method aimed at investigating contemporary phenomena in their natural context [15]. The case study reported in this paper aims to identify and categorize the origins of logical dependencies in the GW project by mining its SVN repository. Our motivation primarily derives from previous studies that have stressed the interplay between logical dependencies and software quality [7, 8, 9, 10, 11, 12].

In the next subsections, we present the case study design and planning. We describe the characteristics of the case, the rationale for choosing this particular case, the data collection instruments and methods, the data filtering techniques used and their reasons, and the data analysis instruments and methods.

3.1 The case study

We conducted an *explanatory case study*, in which the researcher seeks an explanation of a situation or a problem [16]. In contrast to embedded case studies, where multiple units of analysis are studied within a case, our case study is essentially holistic, i.e. the case is studied "as a whole".

According to Seaman [20], a combination of quantitative and qualitative data (a.k.a., "mixed methods" [15]) often provides deeper understanding of the phenomenon of interest. Our case study relies on both quantitative data (file types, size of revisions, developers' contribution level, and logical dependency metrics) and qualitative data (revision comments, code diffs, and informal interviews with the developers). In particular, the quantitative data was primarily employed to filter out the qualitative data to be subsequently analyzed.

3.1.1 The case choice

For the case study, we needed a software project that satisfied the following requirements: (i) open-source software hosted on a SVN repository with anonymous read access, since our mining tool only supported SVN (see Section 3.2); (ii) availability of its

developers to gather information about the project during the process of classification of logical dependencies; (iii) rather small version history (500 to 1k revisions), as this study assumes a manual classification of logical dependencies; and (iv) code written in an object-oriented language (such as C++, C#, or Java), as this would enable us to investigate reflection and dependency injection techniques in the context of logical dependencies.

We selected the Groupware Workbench (GW) software, which is a FLOSS project developed in partnership between three Brazilian universities: USP, UFES, and PUC-Rio. GW satisfies all of the previously stated requirements, since (i) it is hosted at Google Code and stored in a SVN repository; (ii) it is highly active and most developers remain at the university working on it; and (iv) it is entirely written in Java. Regarding (iii), we decided to analyze only the project's trunk folder in SVN, since it accounted for 727 (47%) of all 1541 project revisions. This analysis relies on all GW history in SVN, which corresponds to a development period of two years and three months.

The GW project is supported by the IME-USP FLOSS Competence Center and its project manager is one of the authors of this study. This author was not interviewed for the purposes of this paper. GW is written in Java (approximately 40k lines of code) and it aims to offer a component-based toolkit for the development of Web 2.0 collaborative systems. GW consists of two parts: the kernel and the component kits. The workbench kernel (*component frameworks*) supports the installation, update, grouping, customization, reuse, and life cycle management of the components. The component kits support the development of collaborative tools. The components are manipulated by file system operations and customized by descriptor files. A mobile version of GW is being developed for Android-based devices. Currently, GW is being used in the development of three projects: a social network site focused on the sharing of Brazilian architecture images, a collaborative platform for news publishing, and an online FAQ about FLOSS software in general.

3.2 Data collection instruments and methods

Empirical studies that mine software repositories usually require extensive tool support due to the large and complex data that need to be collected, processed, and analyzed [21]. In this study, we employed XFlow [19], an extensible interactive tool we have developed and whose general goal is to provide a comprehensive analysis of software projects evolution by mining software repositories and taking into account both technical and social aspects of the development process. As illustrated in Figure 2, XFlow collects and parses the log messages of revisions from version control systems (data collection phase), identifies dependencies from the extracted data (processing phase), evaluates metrics over the collected data (metrics phase), and finally presents interactive visualizations depicting the entire software project (visualization phase). Currently, only SVN repositories are supported by XFlow.

In this study, we initially used XFlow to obtain GW basic information, such as the size of revisions, developers' contribution level, and the distribution of file types. After that, we used the tool to identify logical dependencies from source code files in the GW repository.

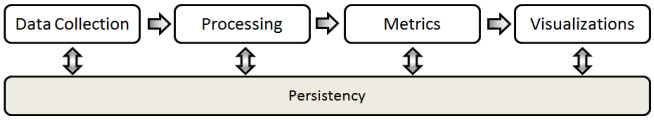

Figure 2. XFlow processing phases

Based on initial interviews with GW developers, we identified the need to implement a modified version of the original sliding time window algorithm [22], since some developers reported checking-in code related to the same task in different (but also close) moments. The sliding time window is a technique proposed by Zimmerman and colleagues [22] to reconstruct change transaction in VCSs that do not support atomic commits (most notably, CVS). This algorithm, which is an improvement over the *fixed time window* algorithm, restricts the maximal gap between two subsequent commits of a transaction (Figure 3), i.e. the beginning of the time interval is shifted to the most recent commit. We adapted the sliding time window algorithm to group SVN revisions from an author within a specified time window and implemented it on XFlow. This procedure refined the process of logical dependencies identification, in the sense that new dependencies between files were captured. To the best of our knowledge, this is the first time that the algorithm has been adapted and applied to an atomic commit featured VCS, such as SVN.

In interviews with GW developers, they recommended a time window of 2 to 3 minutes. Values from the literature [12, 21] converge to a window of 200 seconds. Therefore, based on both the information provided by GW developers and previous work, we decided to apply a time window of 200 seconds.

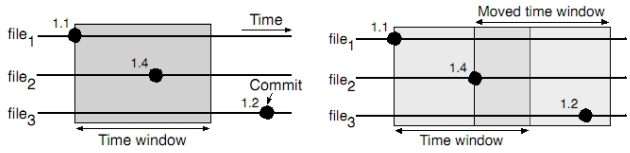

Figure 3. Sliding time window algorithm [21]

To cope with possible remote repository instability, we built a local mirror of the GW SVN repository. After mirroring the repository, we executed the data collection and processing phases of XFlow (Figure 2), which resulted in the identification of "raw" logical dependencies. The method used by the tool relies on the construction of the *task dependency matrix* proposed by Cataldo and colleagues [9, 23]. Finally, we used the tool again to apply the sliding time algorithm and to obtain the new set of logical dependencies derived from grouped revisions. The results of data collection are given in Section 4.1.

3.3 Data filtering

Support and confidence thresholds are commonly used to filter logical dependencies and it is done in a project to project basis [12, 22]. We applied a simple statistical analysis to choose appropriate threshold values for our study. We first analyzed the confidence measure, which gives the strength of the logical dependencies (association rules). The goal was to filter out dependencies whose value of confidence was too low based on a cumulative line graph. Subsequently, we analyzed the support measure for the remaining logical dependencies. Since we intended to perform a manual analysis of dependencies, we wanted to limit the number of logical dependencies based on their relevance. Therefore, we performed a quartile analysis on the distribution of the *number of logical dependencies per support*

value variable and selected only dependencies whose support value was high (outlier values). The results of data filtering, as well as the chosen thresholds, are given in Section 4.2.

3.4 Data analysis instruments and methods

We created a spreadsheet containing the antecedent (LHS) and consequent (RHS) files of every logical dependency. Afterwards, for each dependency, we identified all associated revision numbers and authors' comments. All this information was obtained by querying the project's database generated by XFlow.

Data triangulation, which involves taking different angles of observation towards the studied object, is an important technique used to increase the precision of empirical research [17]. The conclusions regarding the origins of logical dependencies were reached based on such technique, since we counted on (i) the revision comments, (ii) the 7-year software development experience of the first author of this study to examine code that is changed together, and (iii) the feedback from GW active developers. In particular, for every non-trivial classification, we involved GW developers and asked for their support. The results of data analysis are given in Section 4.3.

4. RESULTS

In the next subsections, we present the results of data collection, the results of data filtering, and the classification of logical dependencies' origins.

4.1 Results of Data Collection

In the next subsections, we present information related to the size of revisions, developers' contribution levels and file types distribution in GW.

4.1.1 Revisions Size

The size of revisions provides insights about commit habits in the project. We analyzed the size of project revisions by calculating basic descriptive statistics for the *number of files per revision* variable (see Table 1 and Table 2). A graphical summary showing the variable distribution (A) and the associated boxplot (B) is given in Figure 4.

Table 1. Number of Files per Revision – Descriptive Statistics

N	Sum	Mean	StDev	Skewness	Kurtosis
692.0	9,536.0	13.78	43.16	8.16	83.75

Table 2. Number of Files per Revision – Quartile Analysis

Min	Q1	Median	Q3	Max	IQR	Up. Whisker
1.0	1.0	3.0	10.0	612.0	9.0	23.0

Number of revisions and number of files. Executing the first phase of data analysis resulted in 692 grouped revisions (out of the 727 initial revisions). The sum of number of files per revision was 9,536 files.

Mean and Standard deviation. The mean value indicates that revisions contain approximately 14 files in average. Standard deviation value shows that the dispersion is high, i.e. there were revisions that involved very few files and there were revisions that involved a reasonably large number of files. The following descriptive statistics measures were employed to better understand this dispersion value.

Skewness and Kurtosis. The positive skewness value (8.16) indicates that the data-set is right-skewed, i.e. the tail of the distribution points to the right. The high kurtosis value (83.75) indicates that the data-set has a distinct peak near the mean, declines rather rapidly, and has heavy tails.

Test for normality. Analysis of skewness, kurtosis, and frequency histogram showed that data is not normally distributed. In fact, we conducted the Kolmogorov-Smirnov normality test and obtained a p-value < 0.010.

Quartile analysis. The boxplot gives another view on the distribution of the data-set, by showing its shape, central tendency, and variability. Since the interquartile value is 9, the lower and upper whiskers reveal that "usual" revisions encompass 1 to 23 files. The largest revision in GW included 612 files, which corresponds to GW version 0.1 being moved from a branch to the trunk folder of SVN.

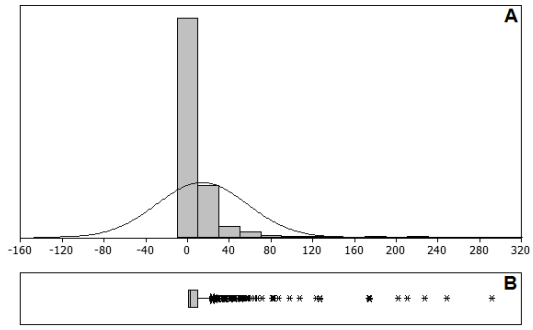

Figure 4. Graphical summary of *number of files per revision*

4.1.2 Developers' Contribution

We computed the *number of worked files per developer* (Figure 5) and the *number of commits per developer* (Figure 6) in order to identify main developers whom we would interview during the study.

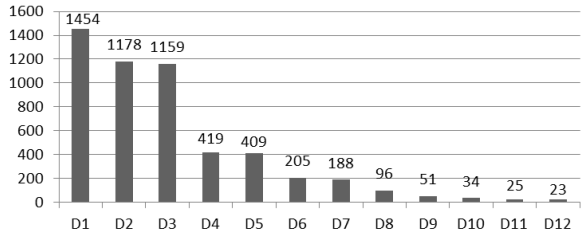

Figure 5. Number of worked files per developer

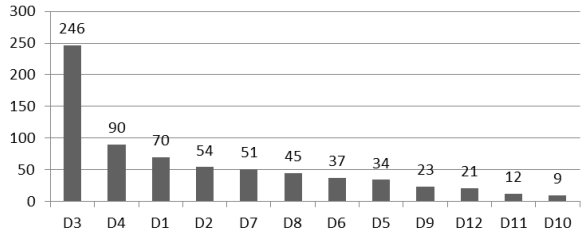

Figure 6. Number of (grouped) commits per developer

Developers D1 and D2 have already left the project and were not available anymore, thus they have been excluded from our analysis. Developers D3 and D4 were our main contacts, since they worked with a high number of files (1159 and 419, respectively) and performed the highest number of commits (246

and 90, respectively) throughout the development of GW. In particular, D3 is responsible for approximately a third of the total number of commits.

4.1.3 File Types

In order to recognize the artifacts that composed the GW throughout its development, we computed the distribution of file types (number of files with a particular file type) by mining its SVN repository with XFlow (Figure 7).

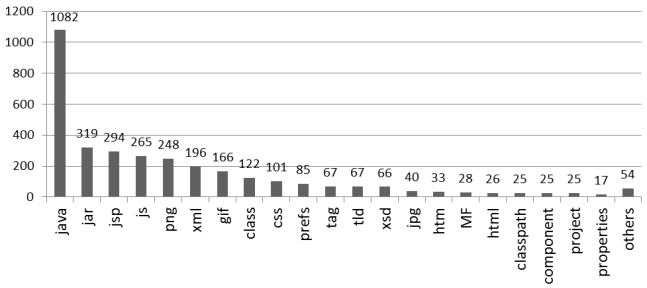

Figure 7. File type distribution

Approximately 33% of all files are Java classes. Other relevant file types include jar, jsp, js, png, xml, class, css, prefs, tag, tld, and xsd.

4.2 Results of Data Filtering

We used the XFlow tool and it identified 1,237,128 logical dependencies (based on grouped revisions) in GW. As stated in Section 3.1.1, we analyzed the project's trunk folder only, which accounts for 727 revisions. In this section, we report the results of the application of the data filtering techniques described in Section 3.3.

Setting a threshold for the confidence measure. We set the threshold for the confidence measure by evaluating the cumulative confidence line graph in Figure 8. The horizontal axis denotes the confidence values, while the vertical axis denotes the cumulative percentage of covered logical dependencies. We concluded that confidence values greater than or equal to 50% covered approximately 78% of all logical dependencies. Therefore, we selected logical dependencies whose confidence value was at least 50%.

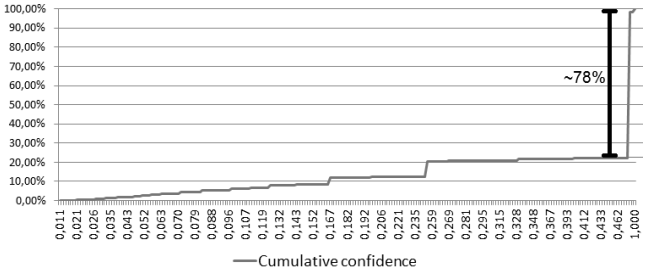

Figure 8. Cumulative confidence

Setting a threshold for the support metric. After applying the confidence filter, we analyzed the remaining set of dependencies to define the support threshold. At first, we excluded logical dependencies with support value equal to 1, as they are clearly not relevant. We then calculated the *number of logical dependencies per support value* (Table 3).

Table 3. Number of logical dependencies per support value

Support	Number of Logical Dependencies
22	1
16	2
15	1
14	3
11	3
9	17
8	35
7	82
6	42
5	100
4	778
3	2801
2	9778

Afterwards, we performed a quartile analysis (Table 4 and Figure 9) in order to select only the outlier values. This guaranteed that the manual inspection would be focused on relevant logical dependencies only. Therefore, after conducting such analysis, we picked only the dependencies whose support value was higher than 4.

Table 4. Support of logical dependencies - Quartile Analysis

Min	Q1	Median	Q3	Max	IQR	Up. Whisker
1.0	2.0	2.0	3.0	22.0	1.0	4.0

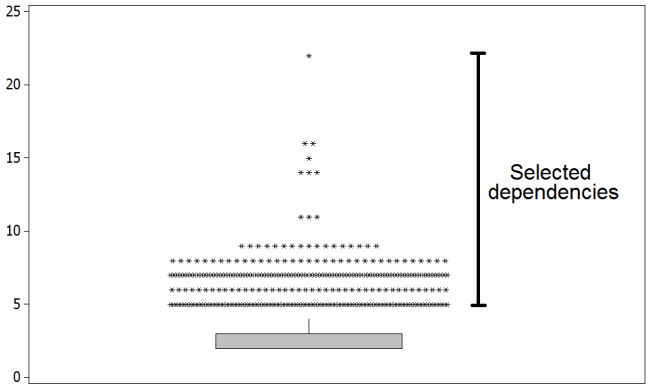

Figure 9. Boxplot for support values

As a final result, we obtained a set of 286 relevant logical dependencies.

Table 5. Logical dependencies per file type

Row #	LHS	RHS	Sup.22	Sup.16	Sup.15	Sup.14	Sup.11	Sup.9	Sup.8	Sup.7	Sup.6	Sup.5	Total	Total%
1	Java	Java	0	1	0	1	1	10	7	72	33	89	214	74.8% (1.83)
2	Java	JSP	0	0	0	0	0	0	0	0	0	1	1	0.3% (0.03)
3	XML	XML	0	0	0	0	2	6	3	0	0	2	13	4.5% (3.40)
4	XSD	XSD	0	0	0	0	0	0	6	3	0	0	9	3.1% (2.10)
5	XSD	XML	0	0	0	0	0	0	9	5	0	0	14	4.9% (10.82)
6	XML	XSD	0	0	0	0	0	0	0	1	0	0	1	0.3% (0.77)
7	XML	Props	0	0	1	0	0	0	0	0	0	1	2	0.7% (6.00)
8	JSP	JSP	0	1	0	2	0	0	2	0	2	3	10	3.5% (0.12)
9	JS	Java	0	0	0	0	0	0	0	0	1	0	1	0.3% (0.03)
10	JS	JSP	0	0	0	0	0	0	0	0	2	0	2	0.7% (0.26)
11	Tag	Tag	0	0	0	0	0	0	0	1	1	4	6	2.1% (1.36)
12	Prefs	Prefs	0	0	0	0	0	1	8	0	0	0	9	3.1% (1.26)
13	Props	Props	1	0	0	0	0	0	0	0	3	0	4	1.4% (14.71)
	Total		1	2	1	3	3	17	35	82	42	100	286	
	Total (%)		0.3%	0.7%	0.3%	1.0%	1.0%	5.9%	12.2%	28.7%	14.7%	35.0%		100,0%

4.3 Logical Dependencies Origins

We began our analysis by investigating the distribution of logical dependencies according to file types of LHS and RHS (Table 5). Values in parenthesis in the last column refer to a normalized total[3] based on file type distribution (Figure 7).

The horizontal lines of the table present the total of logical dependencies according to file types of LHS and RHS. We noticed that approximately three quarters of logical dependencies were established between Java files (row 1). However, in terms of normalized totals, logical dependencies involving Property files (row 13) were the most frequent ones. Also, logical dependencies whose LHS and RHS are of types XSD and XML respectively (row 5), also presented a high normalized total. This last situation seems plausible, since XSD files express a set of rules to which an XML document must conform, i.e. XML files depend on XSD files. Furthermore, we noticed that the established logical dependencies involve only a small subset of GW file types.

The columns of Table 5 present the number of logical dependencies per support value. The first top five support values have a similar number of logical dependencies. In particular, this support interval accounts for only 3.5% of all logical dependencies. Therefore, we conclude that there are a small number of highly logically coupled files in the system. Interestingly, this is also true for Java logical dependencies. The other five support values have a much higher number of logical dependencies. In particular, the lowest support value roughly accounts for one third of the logical dependencies.

In the next subsection, we discuss the preliminary results of the manual inspection of logical dependencies origins.

4.3.1 Manual Inspection

We manually investigated the origins of 75 logical dependencies, which correspond to approximately one quarter of the total number of relevant logical dependencies (Section 4.2). These logical dependencies encompass all GW Java classes from the reflection package `(.../commonswidgets/reflection)`, the upload package `(.../communic/upload)`, and database package `(.../bd/jpa/entities)`. Dependencies among some Java test classes also took part in the analyzed set. As stated in Section 3.4, the manual investigation comprised evaluating revision comments and code diffs, as well as holding informal interviews with the developers.

From the results of our inspection, we concluded that the established logical dependencies involved files that *changed together for different reasons*. A real example involving a logical dependency with a support value of 9 is shown in Table 6.

Table 6. Real example of logical dependency in GW depicting different reasons for joint change

LHS		**RHS**
…/UploadMgrInstance.java		… /CommentMgrInstance.java
Joint-change	**Revision**	**Origin of change**
1	1172	Java packages renamed
2	1186	Applying software license to Java files
3	1203	Structural dependency on a third element
4	1220	Refactoring elements pertaining to a same semantic class
5	1224	Refactoring elements pertaining to a same semantic class
6	1245	Refactoring elements pertaining to a same semantic class
7	1307	Annotations package created
8	1507	Changes in header of Java files
9	1508	Changes in header of Java files

Based on the individual analysis of 408 joint-changes, we conceived the categorization listed in Table 7.

[3] `Total*10000/(type(LHS)*type(RHS))`, when file types of LHS and RHS are different.

`Total*10000/(type(LHS)*(type(RHS)-1))`, otherwise.

`type({LHS, RHS})` refer to the number of files whose type is equal to that of LHS or RHS.

Table 7. Origins of joint-changes

Category	Joint-changes	Total %
Refactoring elements that belong to a same semantic class	80	19.6%
Structural dependencies on a changing semantic class	9	2.2%
Cross-cutting concerns	165	40.4%
Overloaded revision	60	14.7%
Repository operations	21	5.1%
Structural dependencies on specific elements	66	16.2%
Other reasons	7	1.7%
Total	408	

In the following, we describe each category, providing illustrative examples, and pointing research opportunities.

Refactoring elements that belong to a same semantic class. We noticed that artifacts changed together due to refactoring actions made upon a semantic class. We denote by *semantic class* the group of software artifacts that intrinsically share a same basic functionality or architectural role (e.g., entity classes, test classes, controller classes, and persistency layer classes). In GW, some actions that resulted in this kind of joint change were, for instance, changing the default runner of Java test classes and including a specific method in all controller elements. Therefore, we believe that designing software architectures where semantic classes are easily identifiable should improve software evolvability. Moreover, identifying such semantic classes should also enable and support planned maintenance, since the set of naturally impacted classes would be known prior to applying a change.

Structural dependencies on a changing semantic class. This category is a special case of the previous one. The logical relations are characterized as a side-effect of refactoring actions made upon a semantic class. An illustrative example is given in Figure 10, which depicts four classes that changed together in a specific revision.

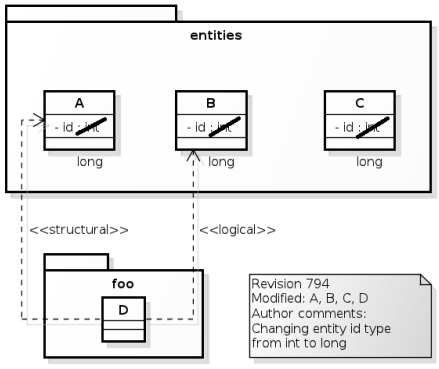

Figure 10. Example of a structural dependency on a changing semantic class (entity)

Classes `A`, `B`, and `C` belong to a semantic class (entity) that went through a change (changing `id` attribute type from `int` to `long`). Although class `D` structurally depends on class `A` only, the logical dependency involving files `D` and `B` (or `D` and `C`) gains one joint-change, since these two files also changed together. Hence, structural dependencies on an element belonging to a semantic class potentially originate different logical dependencies.

Cross-cutting concerns. Cross-cutting concerns refer to non-core concerns (e.g., logging, transaction management, concurrency control) that are spread among a significant amount of modules of a software system (Figure 11).

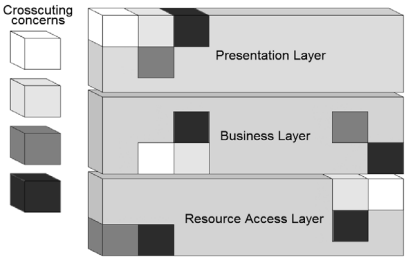

Figure 11. Cross-cutting concerns in a software system

Examples found in GW include applying a software license in Java files, changing the header of Java files, and implementing the Java `Serializable` interface (for saving and restoring the current state of an object to a stream). This provides some evidence that a cross-cutting concern tends to form logical dependencies among the large number of elements that rely on such concern.

Therefore, we believe that examining logical dependencies may serve as an effective way to identify cross-cutting concerns that can be further encapsulated into aspects [24, 25] in order to improve system modularity [26]. In fact, this category corroborates the results found in [27, 28].

Overloaded revisions. Surprisingly, we noticed that pairs of files changed together simply by chance or by convenience. We identified the particular situations:

(i) Multi-action revisions. This occurs when an author modifies different files for different reasons and commits them all together, as illustrated in Figure 12. Occasionally, an author also performs actions that are not explicitly documented in the revision comments. Therefore, during our analysis, revision comments only served as general guidelines for identifying the reasons of joint-changes.

This kind of revision leads to the establishment of unexpected logical dependencies between files. A real example is given by revision 1276, which incorporated six completely different actions: improving three non-related classes, fixing a bug, excluding a jsp page, implementing transaction support, creating a test class, and changing the order of tasks in ant scripts.

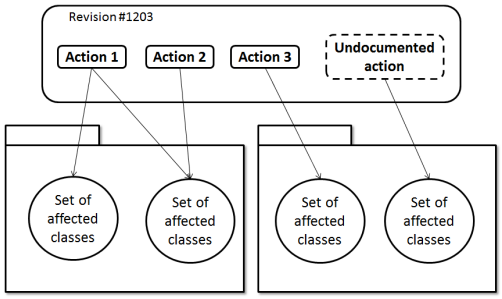

Figure 12. Multi-action revision

(ii) Convenience. Under some circumstances, a class is changed simply by convenience. For instance, while a developer was searching for entity classes to have their id type changed from `int` to `long`, he came across a class whose code was not well-formatted (bad layout). Therefore, he decided to fix the formatting

of such class. Another example included a developer fixing text encoding of a class while implementing the `Serializable` interface in all appropriate classes.

Hence, the joint-changes resulting from overloaded revisions (including those with undocumented actions) leaded to the establishment of "fake" logical dependencies among files, which hinders the effectiveness of maintenance techniques and tools based on this kind of dependency (such as file change prediction [12], or defining coordination requirements among developers [23]). By inspecting logical dependencies, we concluded that the high average number of files per revision (13.78) in GW partially derives from these overloaded revisions.

Mechanisms could be developed for measuring the degree of "overloadness" of revisions. A naïve approach could consist in counting the number of periods ('.') in comments in order to recognize the number of different actions taken by the author.

Repository operations. Repository operations usually involve moving a large number of files across folders. In revision 722, 327 files were moved from a branch to the trunk folder of the project in SVN. Although only one repository operation was identified in GW, it generated a great number of joint-changes and contributed to the establishment of logical dependencies (since many pairs of files ended up being changed together at least once).

Structural dependencies on specific elements. Software artifacts changed together due to structural dependencies from clients to specific suppliers. Although this phenomenon was somehow expected, non-structurally related subclasses also changed together due to a replacement of their corresponding supertype class. Architectural changes, like reorganizing classes in new packages and renaming existing packages, also contributed to the establishment of logical dependencies among the affected elements.

Classic Software Engineering literature has long stated that structural coupling should be minimized because every time a supplier class changes, its clients are also likely to change [26, 29, 30]. Interestingly, only a small amount of joint-changes (16.2%) was directly associated with structural dependencies. This corroborates the results of previous work on the topic [31].

Other reasons. We noticed that classes changed together due to an internal functionality being implemented in GW. This is different from the "overloaded" category, where files changed together for distinct purposes that are not connected to the implementation of a specific functionality. We also noticed that a few classes changed together because they undergone code formatting.

5. THREATS TO VALIDITY

There are some factors that may have influenced the validity of the study.

Internal validity. The subset of analyzed logical dependencies involved files that changed together for various reasons. In particular, this subset encompassed logical dependencies whose support value was no larger than 9. It is thus possible that logical dependencies with top support values (i.e., 22, 16, 15, 14, 11) may reveal a single distinguished origin for the dependency.

We contacted and interviewed developer D3 in order to gather his impressions against data in Table 5. Developer D3 stated that he expected a higher amount of logical dependencies involving Java and JSP files. We tried to identify these dependencies by mining XFlow repository with different confidence and support thresholds, but we did not succeed. We plan to interview other developers to gather their impressions about this same phenomenon (since developer D3 could be simply wrong). In case the problem is confirmed, we will investigate alternative techniques for grouping transactions, such as those evaluated by Pirklbauer [32].

External validity. Commit habits of developers in GW may have influenced the generalizability of the results of this study. In order to have a baseline to compare to, we computed the same descriptive statistics shown in Section 4.1 for the first 100k revisions of the Apache Software Foundation (ASF) SVN repository[4]. In this repository, the mean number of files per revision is 5.38 (versus 13.78) and "usual" revisions encompass 1 to 6 files (versus 1 to 23). Therefore, in average, GW developers commit much more files per revision than other developers from a random ASF project. As noted in Section 5, this phenomenon was partially explained by overloaded revisions. Although we have carefully analyzed such kind of revisions, other software projects with more focused revisions could possibly yield a smaller number of logical dependencies. In addition, the intrinsic logical relation between LHS and RHS files would be stronger in such projects.

Threats to the generalizability of this study are given by the very nature of the employed research design. McGrath states that research methods can be evaluated on three dimensions (generalizability, realism, and precision) and he argues that no method is able to satisfy all dimensions at the same time. In particular, case studies naturally maximize realism, but seldom fully satisfy generalizability (since they involve a small number of non-randomly selected situations) or precision (because there is a low level of control over influencing factors). Hence, we leverage the realism of our results and conclusions. Nevertheless, given common knowledge in software engineering and current research in mining software repositories area [27, 28, 33], we believe that at least some of the categories listed in Table 7 (such as "cross-cutting concerns", "refactoring elements that belong to a same semantic class", and "repository operations") should occur in other software projects with similar characteristics. Therefore, we also consider that our taxonomy might be used as a basis for a more comprehensive and detailed classification of logical dependencies in other software projects.

6. RELATED WORK

In the following, we discuss related work regarding the unveiling of the origins of logical dependencies, as well as tool support for mining software repositories.

Origins of logical dependencies. Cataldo *et al.* conjectured that logical dependencies origins could be related to cascading function calls, semantic dependencies, and platform evolution. Although the meaning of "semantic dependencies" is not clearly given by the authors, we believe that it might be associated with the categories "refactoring elements that belong to a same semantic class" and "structural dependencies on a changing

[4] ASF has a single SVN repository that hosts all its projects. This repository owns more than 1.1 million revisions.

semantic class" that were conceived during our logical dependencies origins analysis.

Hanakawa studied the relation between sets of *highly structurally coupled elements* (M) and sets of *highly logically coupled elements* (L) throughout time [13]. The hypothesis stated by the author is that the average intersection between M and L tends to decrease throughout time due to an increase in "copy and paste" actions (leading to logical coupling only), and developers forgetting to commit structurally related classes at once (leading to structural coupling only). We plan to run PMD[5] copy and paste detector module on GW and then check whether framed classes originated logical dependencies.

Costa *et al.* developed a tool called RaisAware, which aims at supporting the relationship between software architecture and the coordination of software development activities [14]. While defining logical dependencies (co-changes), the authors stated that the uses of reflection and dependency injection techniques can be detected by logical dependencies analysis. In GW, although we examined the *reflection* package, we were not able to find any joint-change that was caused by reflection mechanisms.

During our analysis, we noticed that the establishment of some logical dependencies was connected to the existence of cross-cutting concerns in the system. In fact, Breu *et al.* developed a mining technique that relies on both formal concept analysis (algebraic theory) and a more specific notion of logical dependencies to identify the introduction of cross-cutting concerns [27]. Adams *et al.* developed a more powerful concern mining technique named COMMIT that addresses three common shortcomings found in related work: the inability to merge seeds with variations, the tendency to ignore important facets of concerns, and the lack of information about the relation between seeds [28].

Tool support for mining software repositories. Zimmerman *et al.* developed a tool called eRose, which is an Eclipse plugin that mines CVS repositories to identify logical dependencies and guide developers along related changes [12]. In eRose, the project preprocessing phase is time-consuming and cannot be interrupted. In turn, XFlow supports incremental preprocessing, which enables the analysis of projects with a large number of commits [31]. Furthermore, XFlow is able to preprocess projects residing on a remote SVN.

As acknowledged by Bevan *et al*. in the development of the Kenyon tool, software repositories offer several challenges for data mining due to the large amount of computational resources required to handle them [34]. To address such problem, XFlow counts on data filters and an efficient data structure to map dependencies. For a more thorough comparison between XFlow and other software evolution supporting tools, we recommend [19].

Other work. Finally, there is also a considerable amount of less related work that employed logical dependencies to different purposes, such as to detect design issues [10] and software instabilities [35], infer code decay [11], predict changes in software artifacts [12], establish coordination requirements among developers [22], and support software evolution exploratory studies [19].

[5] http://pmd.sourceforge.net/cpd.html

7. CONCLUSION AND FUTURE WORK

Cataldo *et al.* suggest that a better understanding of the nature of logical dependencies has implications in diverse areas, such as in software quality and in the enhancement of development tools [9]. In this paper, we have investigated the origins of logical dependencies by means of a case study involving a Java FLOSS project. We conducted a manual inspection of logical dependencies origins by reading revision comments, looking at code diffs, and holding informal interviews with the project developers. Preliminary results showed that there was no distinct underlying reason behind the establishment of the analyzed logical dependencies, since they involved pairs of files that changed together for different reasons. We then conceived a categorization for the joint-changes involved in the establishment of such dependencies. We believe that our approach for logical dependencies identification, grouping, and classification can also be reused and adapted for future research in the same domain.

As future work, we plan to inspect a larger set of dependencies from GW, so that we can broaden our results and conclusions. As stated in Section 5, we also plan to investigate other strategies for grouping transactions [32] and compare them to our modified sliding time window algorithm. We also believe that the proposed taxonomy could be validated in other software projects by conceiving and developing automated mechanisms to check joint-changes against each one of the categories listed in Table 7. Investigating the origins of logical dependencies from other software projects should also improve and extend our taxonomy. We also envision large-scale quantitative experiments aimed at unveiling the origins of logical dependencies. For instance, one could investigate the relation between logical dependencies and file types by analyzing a large quantity of FLOSS projects written in a specific programming language. Finally, investigating the interplay between the different kinds of dependency [31] (e.g. structural [29, 30], logical [5, 6], data-flow/hidden [33]) should be a fertile research topic with implications in software maintenance and evolution areas.

8. ACKNOWLEDGMENTS

We thank Mauricio F. Aniche and Mauricio de Diana for insightful contribution to the design of this study. We thank the anonymous reviewers for their valuable comments. We thank Felipe P. Breda for the text review. Marco Gerosa receives individual grant from CNPq. This work is also partially supported by HP (Baile project), CHOReOS, and FAPESP.

9. REFERENCES

[1] Briand, L. C., Wust, J., Daly, J. W. and Porter, D. V. 2000. Exploring the Relationships between Design Measures and Software Quality in Object-Oriented Systems. *J. Systems and Software*, vol. 51, pp. 245- 273, 2000.

[2] Hutchens, D. H. and Basili, V. R. 1985. System Structure Analysis: Clustering with Data Bindings. *IEEE Trans. Software Eng.*, vol. 11, no. 8, pp. 749-757, Aug. 1985.

[3] Selby, R. W. and Basili, V. R. 1991. Analyzing Error-Prone System Structure. *IEEE Trans. Software Eng.*, vol. 17, no. 2, pp. 141-152, Feb. 1991.

[4] Yau, S. S., Collofello, J. S. and MacGregor, T. 1978. Ripple effect analysis of software maintenance. *Computer Software and Applications Conference, 1978. COMPSAC '78. The*

IEEE Computer Society's Second International, pp. 60- 65, 1978.

[5] Gall, H., Hajek, K. and Jazayeri, M. 1998. Detection of logical coupling based on product release history. In *Proceedings of the International Conference on Software Maintenance* (ICSM '98). IEEE Computer Society, Washington, DC, USA, 190.

[6] Ball, T., Kim, J.M., Porter, A. A. and Siy, H. P. 1997. If your version control system could talk. In *ICSE Workshop on Process Modeling and Empirical Studies of Software Engineering*.

[7] Graves, T. L., Karr, A.F., Marron, J. S. and Siy, H. 2000 Predicting Fault Incidence Using Software Change History. *IEEE Trans. Software Eng.*, vol. 26, no. 7, pp. 653-661.

[8] Mockus, A. and Weiss, D. 2000. Predicting Risk of Software Changes. 2000. *Bell Labs Technical J.*, vol. 5, pp. 169-180.

[9] Cataldo, M., Mockus, A., Roberts, J. A. and Herbsleb, J. D. 2009. Software Dependencies, Work Dependencies, and Their Impact on Failures. *IEEE Trans. Software Eng.*, vol.35, no. 6, pp. 864-878, 2009.

[10] D'Ambros, M., Lanza, M. and Lungu, M. 2009. Visualizing Co-Change Information with the Evolution Radar. *IEEE Trans. Software Engineering*, vol. 99, pp. 720-735, 2009.

[11] Eick, S. G., Graves, T. L., Karr, A. F., Mockus, A., and Schuster, P. 2002. Visualizing Software Changes. *IEEE Trans. Software Eng.*, vol. 28, no. 4, pp. 396-412, Apr. 2002.

[12] Zimmermann, T., Weissgerber, P., Diehl, S. and Zeller, A. 2005. Mining Version Histories to Guide Software Changes. *IEEE Trans. Software Engineering*, pp. 429-445, June, 2005.

[13] Hanakawa, N. 2007. Visualization for Software Evolution Based on Logical Coupling and Module Coupling. *Software Engineering Conference, 2007. APSEC 2007*. 14th Asia-Pacific, pp. 214-221, 4-7 Dec. 2007 doi: 10.1109/ASPEC.2007.36.

[14] Costa, J., Feitosa, R., and de Souza, C. R. B. 2009. RaisAware: uma ferramenta de auxílio à Engenharia de Software. *Scientia*, pp. 12-24, 2009.

[15] Robson, C. 2002. Real World Research. Blackwell, (2nd edition).

[16] Yin, R. K. 2003. Case study research: Design and methods, 3rd ed. London, Sage, 2003.

[17] Runeson P, Höst M. 2009. Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering*. 14(2):131-164.

[18] Gerosa, M. A. and Fuks, H. 2008. A Component Based Workbench for Groupware Prototyping. *Proceedings of the 1st Workshop on Software Reuse Efforts, 2nd Rise Summer School – RISS 2008*, Recife, Brazil.

[19] Santana, F., Oliva, G., de Souza, C. R. B. and Gerosa, M. 2011. XFlow: An Extensible Tool for Empirical Analysis of Software Systems Evolution. *Proceedings of the VIII Experimental Software Engineering Latin American Workshop* (*ESELAW 2011)*, Rio de Janeiro, Brazil.

[20] Seaman, C. 1999. Qualitative methods in empirical studies of software engineering. *IEEE Trans Soft Eng.* 25(4):557-572.

[21] D'Ambros M., Lanza M. and Lungu M. 2009. Visualizing co-change information with the evolution radar. *IEEE Trans. Software Eng*. 2009;35(5):720–735.

[22] Zimmermann T. and Weißgerber P. 2004. Preprocessing CVS data for fine-grained analysis. In *Proceedings of the 1st International Workshop on Mining Software Repositories (MSR 2004)*: 2-6.

[23] Cataldo, M., Wagstrom, P. A., Herbsleb, J.D. and Carley, K. 2006. Identification of Coordination Requirements: Implications for the Design of Collaboration and Awareness Tools. *Proceedings of the 20th Anniversary Conference on Computer Supported Cooperative Work*, ACM, New York, NY, pp. 353-362, 2006.

[24] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C.V., Loingtier J.M. and Irwin, J. 1997. Aspect-Oriented Programming. *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP 1997),* Jyväskylä, Finland 220–242.

[25] Laddad, R. 2003. AspectJ in Action, Practical Aspect-Oriented Programming, Manning Publications Co.

[26] Parnas, David L. 1972. On the Criteria To Be Used in Decomposing Systems into Modules. *Communications of the ACM* 15 (12): 1053–1058.

[27] Breu, S., Zimmermann, T. and Lindig, C. 2006. Mining eclipse for cross-cutting concerns. In *Proceedings of the International Workshop on Mining Software Repositories* (*MSR 2006*): 94-97

[28] Adams, B., Jiang, Z. M. and Hassan A. E. 2010. Identifying crosscutting concerns using historical code changes. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering* (ICSE '10), Vol. 1. ACM, New York, NY, USA, 305-314.

[29] Booch, G. 2007. Object-Oriented Analysis and Design with Applications, 3rd ed. Addison-Wesley, 2007.

[30] Larman, C. 2004. Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development, 3rd ed.: Prentice Hall, 2004.

[31] Oliva, G. A. and Gerosa, M. A. 2011. On the Interplay between Structural and Logical Dependencies in Open-Source Software. In *Proceedings of the 25th Brazilian Symposium on Software Engineering (SBES 2011)*, São Paulo, Brazil.

[32] Pirklbauer, G. 2010. Empirical Evaluation of Strategies to Detect Logical Change Dependencies. *SOFSEM 2010: Theory and Practice of Computer Science*. pp. 651–662.

[33] Vanciu, R. and Rajlich, V. 2010. Hidden Dependencies in Software Systems. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM2010)*, Timisoara, Romania.

[34] Bevan, J., Whitehead, J. E., Kim, S. Jr. and Godfrey, M. 2005. Facilitating Software Evolution Research With Kenyon. In *SIGSOFT Softw. Eng. Notes* 177-186.

[35] Bevan, J. and Whitehead, J. E. 2003. Identification of Software Instabilities. In *Proceedings of the 10th Working Conference on Reverse Engineering* (WCRE '03). IEEE Computer Society, Washington, DC, USA.