Empir Software Eng
2012, DOI: 10.1007/s10664-012-9225-9

On the Impact of Software Evolution on Software Clustering

Fabian Beck · Stephan Diehl

Published online: 29 August 2012

Abstract The evolution of a software project is a rich data source for analyzing and improving the software development process. Recently, several research groups have tried to cluster source code artifacts based on information about how the code of a software system evolves. The results of these evolutionary approaches seem promising, but a direct comparison to traditional software clustering approaches based on structural code dependencies is still missing. To fill this gap, we conducted several clustering experiments with an established software clustering tool comparing and combining the evolutionary and the structural approach. These experiments show that the evolutionary approach could produce meaningful clustering results. While the traditional approach provides better results because of a more reliable data density of the structural data, the combination of both approaches is able to improve the overall clustering quality. A review of related studies shows that this approach of combining dependency information is also successful in other software engineering applications.

Keywords Software clustering · Software evolution · Empirical study · Code dependencies

1 Introduction

Software clustering is an important discipline in reverse engineering and software maintenance. It deals with the automatic unsupervised grouping of soft-

Fabian Beck · Stephan Diehl
University of Trier
54286 Trier
Germany
E-mail: {beckf,diehl}@uni-trier.de

ware artifacts like functions, classes, or files into high-level structures like packages, components, or sub-systems based on the similarity of the artifacts. Software clustering is applied, for instance, to understand complex software systems (Mancoridis et al, 1999), to restructure software architectures (Anquetil et al, 1999), to identify reusable components (Maarek et al, 1991), or to detect misplaced software artifacts (Vanya et al, 2008).

Software clustering algorithms rely on characteristic information about the software artifacts to compute a pairwise similarity measure and to finally yield a reasonable clustering. Common clustering approaches retrieve this information directly from the static source code in form of structural dependencies based on, for example, method invocations and variable references among methods (Andritsos and Tzerpos, 2005; Mancoridis et al, 1999; Maqbool and Babri, 2007), or inheritance, aggregation, and method invocations among classes (Mitchell and Mancoridis, 2007; Wierda et al, 2006). Some approaches try to improve the clustering by taking dynamic code dependencies recorded during the program execution into consideration (Gargiulo and Mancoridis, 2001; Xiao and Tzerpos, 2005). Other approaches use the source code only indirectly by analyzing variable names and comments (Kuhn et al, 2005). Although there exists such a variety of data sources, only few approaches integrate several of them into their clustering technique (e.g., Andritsos and Tzerpos, 2005; Wierda et al, 2006).

In the last decade, software engineers have become aware of software evolution as an important and largely unused data source to enhance the software development and maintenance process. Information from the evolution of software projects, in particular, information on how developers change the source code, has been leveraged across many applications: It helps project managers to control the development process (Ma, 2008), software architects to detect design flaws (Gall et al, 2003; Zimmermann et al, 2003), developers to find related files or hidden dependencies (Zimmermann et al, 2004), and quality controllers to identify bugs (Kim et al, 2007). But can it also be leveraged to cluster software artifacts?

1.1 Evolutionary Data in Software Clustering

Some research groups have already addressed this question and started to link ideas from both fields of research—software clustering and software evolution. Clustering-centered approaches enrich structural data with some evolutionary aspects: Andritsos and Tzerpos (2005) integrated the file ownership information, a simple evolutionary data source, and improved a clustering approach based on structural data. However, they also found that the integration of the file timestamps, another simple evolutionary data source, tends to decrease the clustering quality. Wierda et al (2006) used the assumption that the intended architecture of a software system is represented in a purer form in the initial version than in a later one. In their case study, combining the current version with the first version by intersecting their structural dependencies improved

the decomposition. Furthermore, Sindhgatta and Pooloth (2007) introduced a transaction-based clustering approach. Adding evolutionary transactions to structural transactions increased the quality of the clustering result.

Other approaches are more evolution-centered and work only on co-change data (files that are often changed together): In one of the first works on software evolution, Ball et al (1997) used a specialized graph layout algorithm on an evolutionary co-change graph. Clusters emerge as visual groups in the graph visualization. Beyer and Noack (2005) refine this approach by revising the graph data structure and the layout algorithm. In both cases the user, however, needs to finally mark the clusters manually. Voinea and Telea (2006) integrate a clustering algorithm based on software evolution into their visualization tool *CVSgrab*. The evolutionary clustering is used to improve the sequential order of the files in the visualization. Vanya et al (2008) are able to identify design flaws in the software architecture by comparing the evolution-based clustering decomposition to the current architecture of the software. In a case study, experts rated most of the detected design flaws as valuable information.

1.2 Objectives

Methods that integrate software clustering and software evolution, such as those discussed above, seem promising. On the one hand, software clustering based on structural data might be improved by integrating evolutionary data sources and, on the other hand, it appears to be possible to cluster software only by using evolutionary data.

But despite these positive results, there are some aspects that are not covered sufficiently yet: The clustering-centered approaches first of all present new clustering techniques and only secondarily employ evolutionary data. The evolution-centered approaches just show that clustering based on evolutionary information is working to some degree, but are not contrasted to approaches based on structural data.

Our goal is to overcome these shortcomings by directly comparing structural and evolutionary data sources to each other. We use an established software clustering approach to recover the architecture of ten open source Java software projects based on three different kinds of dependencies: static structural source code dependencies, evolutionary co-change dependencies, and combined structural and evolutionary dependencies. Different filtering setups and combination strategies lead to a total of 152 different concrete dependency graphs per project. For each of these graphs, we compute a clustering and assess its quality by measuring its similarity to a reference decomposition.

The main contributions of this paper are:

- It presents the first systematic comparison of structural and evolutionary data for software clustering.
- The study is one of the most extensive studies in software clustering with respect to the number of analyzed projects and project sizes.

- The paper describes the first approach that consequently integrates evolutionary information into a traditional clustering technique.
- The paper includes an assessment of the influence that filtering evolutionary data has on the clustering quality.
- We review related approaches and discuss the results of our study in this broader context.

The rest of this paper is organized as follows. Section 2 introduces an experimental design that clusters software systems and measures the quality of the resulting software decompositions. Section 3 presents the study, consisting of three experiments. Finally, Section 4 discusses the validity of the results and Section 5 compares them to the results of related studies. Section 6 concludes the findings.

This work is an extended version of a paper presented at the *Working Conference of Reverse Engineering 2010* (Beck and Diehl, 2010a), which we broadened in three directions: First, we increased the number of analyzed projects from six to ten carefully selected projects (Section 2.4). Second, results are discussed in greater detail, which provides further insights (Section 3.4). Third, a review of related studies enables consolidating our results in a broader context (Section 5).

2 Experimental Design

Our study concentrates on grouping Java classes into packages. Classes are the elementary units in the design process of object-oriented software systems. Their organization into packages reflects the architecture of a software system. Since interfaces are similar to classes, we handle them like classes in the experiments and therefore use the term *class* interchangeable for classes as well as for interfaces.

2.1 Data Sources

In the following experiments the dependency information is the independent variable. A directed graph, where nodes represent classes and edges represent dependencies, models this asymmetric dependency information.

2.1.1 Structural Dependencies

Structural static source code dependencies (short: structural dependencies) are the most widely used data source for software clustering. As such, they represent the conventional approach and constitute the control group in our study. We incorporate all main types of structural class dependencies, namely inheritance, aggregation, and usage (e.g., method calls, method parameters, local variables).

Definition 1 Let $C(S)$ be the set of classes of a software system S. We define four directed relations on classes:

$$(c_1, c_2) \in E_{\mathrm{CIG}} \Leftrightarrow c1 \text{ extends } c2$$
$$(c_1, c_2) \in E_{\mathrm{CAG}} \Leftrightarrow c1 \text{ aggregates } c2$$
$$(c_1, c_2) \in E_{\mathrm{CUG}} \Leftrightarrow c1 \text{ uses } c2$$
$$E_{\mathrm{SCDG}} := E_{\mathrm{CIG}} \cup E_{\mathrm{CAG}} \cup E_{\mathrm{CUG}}$$

The directed graphs $G_x := (C(S), E_x)$ are called Class Inheritance Graph (CIG), Class Aggregation Graph (CAG), Class Usage Graph (CUG), and Structural Class Dependency Graph (SCDG).

These graphs only consider direct dependencies—transitive closures or the like are not computed. To retrieve the graphs, we use *DependencyFinder*[1], a code analysis suite that works on *Java* bytecode and, among other things, is able to extract all relevant dependencies.

2.1.2 Evolutionary Dependencies

The evolution of a software project is documented by the changes applied to its source files in the course of development. In modern software engineering, a revision control system (version archive) such as *CVS* or *SVN* stores these changes. Transactions—changes simultaneously submitted to the version archive by the same developer—are the elementary units in these systems.

Class A depends on class B by evolution if class A has often been changed together with class B. In other words, both classes have often been part of the same transactions. This is the basic idea behind evolutionary dependencies (also referred to as evolutionary couplings or co-change couplings).

Interpreting evolutionary dependencies is based on the assumption that transactions implicitly group dependent files together. But some transactions might relate files randomly, for example, if a developer fixed two totally unrelated bugs in a single transaction. Hence, a mechanism that allows filtering out such noise and considers only strong dependencies might improve the reliability of the evolutionary data.

Zimmermann et al (2003) introduce the concept of support and confidence to measure the strength of evolutionary dependencies. The support value of a dependency counts how often the two software artifacts were changed together. Additionally, the confidence value of a dependency relates the support to the total number of changes applied to one of the artifacts.

Definition 2 Let $c_1, c_2 \in C(S)$ be two classes and $\{T_i\}_{i=1}^{l}$ a sequence of transactions.

$$\mathrm{Supp}(c_1, c_2) := |\{T_i : c_1, c_2 \in T_i\}|$$

[1] `http://depfind.sourceforge.net/`

is called **support** of the evolutionary dependency of class c_1 to class c_2.

If c_1 is element of at least one transaction T_i, the **confidence** of the evolutionary dependency of class c_1 to class c_2 is defined as

$$\mathrm{Conf}(c_1, c_2) := \frac{\mathrm{Supp}(c_1, c_2)}{\mathrm{Supp}(c_1, c_1)}$$

Otherwise, $\mathrm{Conf}(c_1, c_2) := 0$.

In the definition of Conf, the value of $\mathrm{Supp}(c_1, c_1)$ represents the total number of transactions c_1 is part of. Thus, the value of Conf reaches its maximum 1 if c_2 is changed whenever c_1 is changed. Note that Supp is a symmetric function whereas Conf is not. Conf can be interpreted as the conditional probability that c_2 is changed if c_1 is changed.

Analogously to the structural dependencies, an evolutionary dependency graph can be defined with the help of the Conf and the Supp functions. In contrast to the structural graphs, the evolutionary graph depends on two parameters that filter out weak dependencies: a confidence threshold α and a support threshold k. As part of our experiments, we have to find a reasonable setting for these parameters.

Definition 3 Let $C(S)$ be the set of classes of a software system S. We define a directed relation of classes

$$(c_1, c_2) \in E_{\mathrm{ECDG}_\alpha^k} \Leftrightarrow \mathrm{Supp}(c_1, c_2) > k \ \wedge \ \mathrm{Conf}(c_1, c_2) > \alpha$$

The directed graph $G_{\mathrm{ECDG}_\alpha^k} := (C(S), E_{\mathrm{ECDG}_\alpha^k})$ is called **Evolutionary Class Dependency Graph** (ECDG_α^k) with parameters $\alpha \in [0, 1)$ (confidence threshold) and $k \in \mathbb{N}_0$ (support threshold).

We use the approach by Zimmermann and Weißgerber (2004) to extract the evolutionary class dependency graphs from the version archives.

As common when mining software repositories, we omit large transactions to reduce noise in the evolutionary dependency data—transactions with more than 25 classes, or 50 classes respectively for projects already investigated in Beck and Diehl (2010a). The converter also ignores classes from branched versions to avoid conflicts caused by multiple copies of the same class. Refactorings are not tracked; only classes available in the latest version are considered.

Although it is possible to relate non-source files with the concept of evolutionary dependencies, in the context of this study we restrict it to source files to guarantee the comparability to structural dependencies. In real world applications, the possibility to cluster non-source files might be a crucial advantage of evolutionary dependencies over structural dependencies.

2.2 Clustering Algorithm

In general, a clustering algorithm divides a set of entities—here, classes—into clusters. The result of this division is called clustering decomposition. Depending on the algorithm, the decomposition is either flat or hierarchical. Various software clustering approaches have been proposed and studied (e.g., Maqbool and Babri, 2007, provide an overview). Since this work does not aim at improving a particular clustering algorithm directly but at assessing the quality of different data sources, any established software clustering algorithm would serve as an example; the results, however, are only directly valid for the selected approach.

Bunch (Mancoridis et al, 1998; Mitchell, 2002) is a graph-based clustering tool that follows the concept of low coupling and high cohesion (Stevens et al, 1974). Its clustering algorithm optimizes a clustering quality metric with a heuristic search technique and produces hierarchical clustering decompositions. Several evaluations showed that *Bunch* is among the best currently available software clustering tools (Andritsos and Tzerpos, 2005; Maqbool and Babri, 2007; Wu et al, 2005). Since *Bunch* is customizable, we tried to find a good parameter setting. The resulting setup is similar to the ones used in the evaluations cited above.

Bunch works on a graph structure that is called *Module Dependency Graph*. The graph represents modules as nodes and module dependencies as directed edges. Since the terms *module* and *dependency* are not bound to a strict definition, we are allowed to consider the weighted dependency graphs defined in Section 2.1 as *Bunch Module Dependency Graphs*.

Bunch provides three optimization strategies: an exhaustive search algorithm, a hill climbing algorithm, and a genetic algorithm. We prefer the hill climbing algorithm because it produces stable high quality results efficiently in predictable runtime. When using the hill climbing algorithm, further parameters need to be set. Based on some performance tests with the *JFtp* project and a study by Mitchell and Mancoridis (2007), we set the initial population size to 1, chose the nearest ascent hill climbing option, and deactivated further algorithm extensions.

Since *Bunch* uses a heuristic search approach that has a random element, the clustering process can be considered a random experiment. Mitchell and Mancoridis (2007) showed in their study about *Bunch* that in most cases the resulting decompositions differ only slightly. Nevertheless, we performed several repetitions of our experiments to increase the reliability of the results. Due to these repetitions and some performance problems of *Bunch*, we had to restrict our experiments to less than 1000 classes per project.

2.3 Evaluation Method

To decide which data source performs best, we finally have to measure the quality of the clustering result. But evaluating software clustering results is

challenging: There does not exist an optimal solution to the clustering problem due to different paradigms for designing software decompositions and individual preferences of developers. Hence, every evaluation method will be a heuristic. In this section we motivate our evaluation approach and discuss our design decisions.

According to Maqbool and Babri (2007) the quality of a software decomposition can be evaluated either by considering internal quality criteria (internal assessment), or by comparing it to a reference decomposition (external assessment). But since we vary the input data source (not the clustering algorithm), internal metrics are not applicable: They compare the clustering result to the input data (or are at least biased by the input data) and thus rely on a constant input. For instance, measures based on coupling and cohesion use the dependency graph to determine the quality of a software decomposition (Anquetil et al, 1999). Results derived from different input graphs would not be comparable. In contrast, an external assessment allows a quality measurement independent of the input data. Thus, we are confined to using such an external assessment.

2.3.1 Reference Decompositions

The approach of creating a reference decomposition as a benchmark for an external assessment assumes that a perfect clustering exists, which is represented by the reference decomposition. But since there is no single optimal solution in practice, a reference decomposition can only be an approximation of one of the optimal scenarios.

A reference decomposition can be created by employing external domain experts, who create the decompositions manually, or by using the current, factual architecture of the system created by the developers (e.g., the package structure of an object-oriented system). While domain experts may have a more objective view on the system, the developers know their system best and spend much more time thinking about its architecture. But engaging an expert to modularize a particular system is more expensive than retrieving the factual architecture from the source code because the latter can be automated. Since we analyze ten projects in this study, taking the package structure of the projects as the reference decomposition was the only feasible option.

Working with the factual architecture as the reference decomposition, it is important that the architecture is of good quality. But this does not have to be the case for every project: A few less qualitative reference decompositions could be compensated by other projects. Moreover, even if there is some sort of bias, it is very unlikely that this bias is the same across all 10 software projects that we finally analyze in the study.

Assuming that the developers thoroughly designed their systems, a certain quality can be expected at least on average. To decrease the risk of getting projects with a low quality of the package structure, we look at different quality criteria and discard those projects showing problematic characteristics (Section 2.4).

2.3.2 Similarity of Decompositions

The clustering decomposition that is most similar to the reference is considered the best result. While there is no natural metric that measures the similarity between two decompositions, some heuristics are proposed in the literature.

As Wen and Tzerpos (2004) point out those metrics can be classified into three categories: first, metrics based on the nodes of the dependency graph; second, metrics based on the edges of the dependency graph; and third, metrics based on both, nodes and edges. Again, our options are limited because we vary the dependencies and hence cannot use dependency-based metrics if we want to design a fair experiment.

Tzerpos and Holt (1999) developed a node-based metric, called *MoJo*, that estimates the distance between two decompositions computing the minimal number of *Move* and *Join* operations needed to transform one decomposition into the other. Furthermore, Wen and Tzerpos (2004) introduced *MoJoFM*, a revision of *MoJo* normalized by the decomposition most distant to the reference decomposition. *MoJoFM* ranges from 0, representing the most distant decomposition, to 100, representing a clustering that is completely identical to the reference. It simulates the operations a user would perform to transform one decomposition into the other. The metric is clear and simple to understand. For these reasons we preferred *MoJoFM* over comparable node-based metrics like the *Precision/Recall* metric (Anquetil et al, 1999) or the *Koschke-Eisenbarth* metric (Koschke and Eisenbarth, 2000).

Definition 4 Let $\mathrm{mno}(X, Y)$ be the minimum number of Move and Join operations that is needed to transform a flat decomposition X into a flat decomposition Y. For two flat decompositions A, B

$$\mathrm{MoJoFM}(A, B) := 100 \cdot \left(1 - \frac{\mathrm{mno}(A, B)}{\max_x(\mathrm{mno}(x, B))}\right)$$

(where x is an arbitrary decomposition) is called the **MoJoFM similarity** from A to B.

Hence, the metric is normalized by $\max_x(\mathrm{mno}(x, B))$, which is the worst-case number of operations needed to transform an arbitrary decomposition x into B. Though the worst case needs to be found among all possible decompositions, it is not necessary to compute $\mathrm{mno}(x, B)$ for all decompositions x as Wen and Tzerpos (2004) explain in detail.

As B is the criterion for the normalization, the reference decomposition has to be represented by B. The polarity of the scale is reversed to get a measure of similarity instead of distance. Despite the normalization of *MoJoFM*, one cannot compare the *MoJoFM* values of different sample projects directly because the normalization depends on the structure of the reference decomposition. Only comparisons on the same reference—i.e., on the same project—are valid.

MoJoFM works on flat decompositions, that is, the metric ignores the hierarchical structure of the clustering decomposition and of the package structure.

In contrast, the *END framework* (Shtern and Tzerpos, 2004) takes this hierarchical structure into account and *MoJoFM* can be used as a plug-in for the framework. However, we decided against applying it: *Bunch* produces hierarchical decompositions of different height and we observed a considerable systematic variation of these heights for different data sources. Due to the height balancing mechanism of the *END framework*, this variation could systematically bias our results—a risk that we do not want to take for better considering the hierarchical structure. An alternative is the *UpMoJo* metric (Shtern and Tzerpos, 2007), which incorporates the hierarchy by introducing an additional *up* operation to *MoJo*; but also in this case, the systematic variation in height could introduce an unwanted bias.

For employing *MoJoFM*, we, however, have to transform the hierarchical decompositions into flat decompositions. To transform the reference decomposition, we use the lowest level of the package partition ignoring the package hierarchy. To transform the hierarchical clustering result, we cut the hierarchy on the level where the resulting flat decomposition is most similar to the reference (based on the *MoJoFM* value). This solution avoids noise or a bias caused by too fine- or coarse-grained clustering decompositions.

It can also be helpful to visually compare software decompositions: Beck and Diehl (2010b) developed a visualization approach that contrasts two software decompositions while concurrently giving an overview on the dependencies. For evaluating clustering results, visualization can provide valuable additional insights, but cannot replace a metric because in a visualization results have to be analyzed manually and are liable to the subjective interpretation of the analyst.

2.4 Sample Software Projects

The study is to be conducted on real-world software projects. In particular, we analyze ten open source *Java* projects selected from a list of eighteen projects. The first step was to acquire all necessary data for the initial list of eighteen projects, including the source code, executable versions (bytecode) as well as copies of the version archives.

2.4.1 Initial Set of Projects

Table 1 presents meta-data for the initial list of eighteen software projects: basic information like a brief description and the considered version, details on the size of the project such as the number of packages (#P) and classes (#C), and an outline of the evolution consisting of the type of archive, the analyzed time frame, the number of transactions (#T), and the number of developers (#D). In this extended version of the paper, twelve new projects were taken into consideration (*Checkstyle*–*Wicket*).

Although this set of projects cannot be considered statistically representative of the whole population of software projects, it covers a wide range of

Table 1 Characteristic data of the sample software projects and their repositories; number of packages (#P), classes (#C), transactions (#T), and developer (#D)

Project	Description	Version	#P	#C	Archive	Time frame	#T	#D
Azureus[a]	*BitTorrent* client	2.5.0.4	69	477	CVS	2003/07/10 – 2007/02/14	10665	27
JEdit	text editor	4.2	27	840	SVN	2001/09/02 – 2007/02/12	2190	20
JFreeChart	chart library	1.0.4	54	794	CVS	2001/10/18 – 2007/02/14	2413	5
JFtp	*FTP* client	1.0	7	78	CVS	2002/01/25 – 2003/03/23	210	5
JUnit[b]	unit tests	4.2	16	103	CVS	2002/12/12 – 2007/02/08	673	7
Tomcat[c]	*Java Servlet*	6.0.10	38	561	SVN	2006/03/27 – 2007/03/10	661	13
Checkstyle	coding conventions	5.1	21	261	SVN	2001/06/22 – 2010/02/16	1335	6
Cobertura	test coverage	1.9.4.1	19	99	SVN	2005/02/12 – 2010/03/03	226	6
CruiseControl	cont. integration	2.8.4	27	295	SVN	2001/03/26 – 2010/09/16	1615	10
iText	*PDF* library	5.0.5	24	402	SVN	2007/12/20 – 2010/11/02	817	7
JabRef	*BibTeX* manager	2.6	37	499	SVN	2003/10/16 – 2010/04/14	1348	23
JHotDraw	GUI framework	7.6	65	656	SVN	2006/11/22 – 2011/01/09	302	2
LWJGL	gaming library	2.7.1	27	564	SVN	2002/08/09 – 2011/02/10	1557	11
PMD	code problems	4.2	47	565	SVN	2002/06/24 – 2008/03/26	2041	18
Stripes	web framework	1.5.5	19	238	SVN	2005/09/07 – 2011/01/04	812	7
SweetHome3D	interior design	3.1	8	167	CVS	2006/04/11 – 2011/02/13	1807	1
TV-Browser	program guide	2.7.6	62	485	SVN	2003/04/25 – 2010/12/19	4602	12
Wicket	web framework	1.2.2	86	622	SVN	2004/12/21 – 2006/08/27	3456	12

[a] restricted to `org.gudy.azureus2.core3`
[b] test cases excluded
[c] restricted to `org.apache.catalina`

project types—from user clients and libraries to server applications. The numbers of classes (based on the latest version) give an idea of the project sizes: *JFtp* is the smallest project examined with only 78 classes while *JEdit* is the largest one with 840 classes. Note that for the *Azureus* and *Tomcat* project, we only considered one of their main packages because our experimental setup—in particular, the employed clustering tool *Bunch*—was not able to handle more than 1000 classes efficiently. Moreover, we excluded all test cases from *JUnit* because they are arranged in two large unstructured test packages—a structure that conflicts with the idea of grouping dependent files together.

2.4.2 Selection Criteria

To ensure a certain quality of the reference decomposition, we chose the set of sample projects carefully: We look at project and code quality metrics and exclude those projects that show questionable results for one of the quality metrics. The assumption behind this strategy is that the general project and code quality is correlated with the quality of the package structure. We cannot measure the quality of the package structure directly because metrics for this purpose usually depend on structural dependencies (Melton and Tempero,

2007, provide an overview on graph-property based package design measures). Using those or evolutionary dependencies for the selection of software projects could bias the results of the study towards the respective data source. In particular, we take the following criteria into account:

Package Size: Software development experts agree that the size of a package should not exceed a certain limit—they disagree, however, on the quantity of that limit (Melton and Tempero, 2007). Hansen et al (2011) analyzed 1 141 open source Java projects and found that a vast majority of projects has an average beneath 18 classes per package (Hansen et al, 2011, Figure 8). Since we only want to exclude those projects that are clear outliers, we take this value as a threshold and require not more than 18 classes per package on average.

Comment Ratio: As a predictor of maintainability, the ratio of code comments is an established measure of code quality (Arafat and Riehle, 2009). It is, however, hard to name a fixed target range for this metric because the ratio may vary depending on the programming language and the exact metric definition used. Our strategy is here to simply exclude the three projects with the smallest comment ratios. We measure the comment ratio employing the tool CodeAnalyzer[2].

Success: If the developers managed to implement a successful system, the software design is likely to meet at least a certain standard. Following this rationale, we consider success as an indicator of high design quality. Two perspectives are taken into account: The number of downloads, which we could retrieve for projects hosted at *sourceforge.net*, estimates the impact of the system from an end-user perspective. Additionally, developers and experts are covered by counting the people enlisted as users of the system at *ohloh.net*, a directory of open source systems. To filter out the less successful projects, we set the thresholds to at least 500 weekly downloads at *sourceforge.net* and 50 enlisted users at *ohloh.net*.

Other: In some cases, the nature of the project hints at a certain quality of the package structure. For instance, *JHotDraw* is a project primarily developed as a reference for good software design[3]. Moreover, projects such as development frameworks, extensible plug-in architectures, or libraries are also likely to be designed carefully because other developers will have to rely on a comprehensive structure of the code. We consider these kinds of extra properties as positive quality indicators.

2.4.3 Project Selection

Table 2 documents the selection process of software projects. The columns of the table show the quality metrics as defined above together with the conditions that specify the desired ranges for each criterion. If a criterion is violated for one of the projects, the respective value is printed in bold font. The projects that finally made it into the selection are highlighted with gray background color.

[2] `http://www.codeanalyzer.teel.ws/`

[3] `http://www.jhotdraw.org/`

Table 2 Quality criteria for the initial list of software projects; conditions provide desired ranges for the criteria; violations of the criteria are marked in bold font; considered projects for the study are highlighted in gray

Project	#C/pack.	Comment ratio	Downloads[a]	Users[b]	Other
Condition	≤ 18	> smallest three	≥ 500	≥ 50	
Azureus (Az)	6.9	18%	161 283	374	
JEdit (JE)	**31.1**	29%	7 956	196	plug-in architecture
JFreeChart (JFC)	14.7	45%	3 953	117	library
JFtp	11.1	**14%**	**121**	**0**	
JUnit (JU)	6.4	18%	1 608	1 245	
Tomcat (Tom)	17.4	34%	– [c]	1 360	
Checkstyle (Che)	12.4	45%	763	303	
Cobertura	5.2	**15%**	**481**	102	
CruiseControl (CC)	10.9	37%	729	99	
iText (iT)	16.8	39%	3 635	109	library
JabRef	13.5	23%	4 565	**32**	
JHotDraw (JHD)	10.1	31%	**70**	**2**	reference design, library
LWJGL	**20.9**	38%	4 774	**40**	library
PMD	12.0	**14%**	1 529	220	
Stripes	12.5	42%	**145**	**24**	framework
SweetHome3D	**20.9**	22%	146 016	**5**	
TV-Browser	7.8	26%	8 185	**27**	
Wicket (Wi)	7.2	47%	– [d]	131	framework

[a] weekly downloads from sourceforge.net, retrieved 2012-01-09
[b] ohloh.net users, retrieved 2012-01-02
[c] Tomcat is not hosted at sourceforge.net
[d] moved to apache.org

In particular, our strategy for selecting the projects worked as follows: Projects are considered if none of the condition is violated or if each violation is compensated by an extra quality criterion (referenced in the last column).

This strategy leaves ten of eighteen projects: For instance, we did not consider *JFtp*, one of the six projects of the previous study, due to various violations of quality criteria (comment ratio, downloads, and users). But we did consider *JHotDraw* despite of violations of the two success criteria—they are compensated by strong extra evidence. Although we could not retrieve download rates for *Tomcat* and *Wicket*, we included both projects because we assumed for *Tomcat* that it not only has many users but is also downloaded frequently, and because *Wicket* is a development framework.

3 Study

One of the main use cases of software clustering is architecture recovery. When the architecture of a software system is totally undocumented or the documen-

tation is just outdated, software clustering helps to retrieve the current architectural information. An automatically recovered architecture also supports the developers to redesign a badly structured system.

The following study addresses this problem of architecture recovery. It evaluates the quality of a clustering decomposition in terms of its similarity to the package structure as described in Section 2.3. This approach of retrieving the already documented package structure by a clustering algorithm may sound strange, but is an established assessment method for software clustering algorithms (Mancoridis et al, 1999; Anquetil et al, 1999; Wu et al, 2005). The goal is not to use this procedure in practical application, but to get a measure for the quality of a clustering approach. To this end we need examples where we already have a good reference. Outside our experimental environment, we would of course apply the clustering algorithm to projects or subsystems with no documented structure or an assumed bad structure.

The study consists of three experiments. Experiment 1 contrasts structural and evolutionary dependency graphs in the application of software clustering. Experiment 2 looks at the dependency quality in these graphs to better understand the previously gained results. Finally, Experiment 3 focuses on combining both data sources. Each of the experiments explores different setups of dependency graphs to find the best possible solution.

Table 3 Dependency graph sizes

		Az	*Che*	*CC*	*iT*	*JE*	*JFC*	*JHD*	*JU*	*Tom*	*Wi*
#Nodes		477	261	295	402	840	794	656	103	561	622
#Edges	CIG	279	214	203	226	318	484	465	55	312	599
	CAG	434	90	142	372	691	223	555	49	455	417
	CUG	2269	786	1070	2656	4020	3714	3552	294	2207	3441
	SCDG	2362	810	1177	2698	4117	3937	3668	306	2348	3596
	$ECDG^0_{0.0}$	6218	5178	6264	1466	29372	13922	9270	642	696	16054
	$ECDG^0_{0.2}$	1354	2162	1865	1204	8438	9512	6107	487	502	5434
	$ECDG^0_{0.4}$	727	1003	862	1005	4011	5427	3776	310	293	2736
	$ECDG^0_{0.6}$	345	547	355	784	2117	2478	2126	146	177	1557
	$ECDG^0_{0.8}$	277	378	267	704	1655	2063	1897	105	169	1202
	$ECDG^1_{0.0}$	2330	1378	2000	688	9430	2376	1288	138	84	4984
	$ECDG^1_{0.2}$	650	894	779	652	3289	1991	1043	116	63	2309
	$ECDG^1_{0.4}$	302	518	373	580	1542	1334	582	92	45	1142
	$ECDG^1_{0.6}$	172	303	213	531	888	930	424	71	25	636
	$ECDG^1_{0.8}$	104	134	125	451	426	515	195	30	17	281
	$ECDG^2_{0.0}$	1258	586	936	100	4538	696	388	42	24	2432
	$ECDG^2_{0.2}$	419	401	424	92	1713	588	340	33	23	1221
	$ECDG^2_{0.4}$	193	281	222	56	814	338	222	14	18	609
	$ECDG^2_{0.6}$	99	141	90	20	390	172	129	9	13	281
	$ECDG^2_{0.8}$	51	54	55	10	247	56	75	5	11	128

Table 3 characterizes the dependency graphs of the studied software projects in terms of numbers of nodes and edges. Comparing the simple structural dependency graphs with respect to their edge density, the *Class Usage Graph* (CUG) and the *Structural Class Dependency Graph* (SCDG) are far denser than the *Class Inheritance Graph* (CIG) and the *Class Aggregation Graph* (CAG). Thus, clustering might be harder using only inheritance or aggregation dependencies.

The *Evolutionary Class Dependency Graph* (ECDG) depends on two parameters, the support threshold and the confidence threshold. These two parameters can be considered as filters that are getting stronger (i.e., reducing the number of dependencies) with increasing values. In Table 3, we are able to confirm that the number of dependencies decreases for increasing support values as well as for increasing confidence values. Only the slightly filtered graphs contain extensive dependency information. Since it is hard to define reasonable threshold values in advance, the first two experiments will vary support and confidence systematically. Finding a good filtering setup is a trade-off between dependency reliability and dependency density.

3.1 Experiment 1: Simple Data Sources

The first experiment addresses the question whether it is possible to get meaningful clustering decompositions using only structural or evolutionary data sources: The experiment compares the clustering results for the CIG, CAG, CUG, and SCDG to the ECDG in different filtering setups. Table 4 presents the results of the experiment regarding the *MoJoFM* metric values for each clustering setup. As mentioned in Section 2.2 repeated runs were performed ($n = 50$) and averaged to increase the precision of the quality information.

A precision measure of average values is the standard error $\hat{\sigma}_{\overline{x}}$ (the standard deviation of the mean values). In Table 4 and all following tables containing clustering results, less precise *MoJoFM* values with a standard error of $0.5 \leq \hat{\sigma}_{\overline{x}} < 1.0$ are marked with ' and those with a standard error of $1.0 \leq \hat{\sigma}_{\overline{x}}$ with *. The values that are not marked can be considered as stable across the multiple runs of the clustering algorithm ($\hat{\sigma}_{\overline{x}} < 0.5$). The upper bound of the standard error completes each table. Moreover, we highlight the best overall quality values in light gray and the best project specific ones in gray.

3.1.1 Results for Structural Graphs

Starting with the simple structural dependencies (CIG, CAG, and CUG), the highest *MoJoFM* values, indicating a high agreement with the reference decomposition, are mostly reached with the CUG. This fits with the observation that the CIG and CAG usually do not contain as much information as the CUG. There exist, however, some exceptions: For *JUnit (JU)* the quality of the CIG, CUG, and CAG results are nearly equal although the CUG has much

Table 4 MoJoFM clustering quality

$n = 50,\ \hat{\sigma}_{\overline{x}} \leq 1.7$	*Az*	*Che*	*CC*	*iT*	*JE*	*JFC*	*JHD*	*JU*	*Tom*	*Wi*	***avg***
CIG	30.2	46.0'	54.3	51.1	42.0	35.4	37.2	28.5	32.8	51.2'	**40.9**
CAG	28.6	25.2	30.3	48.0	47.1'	14.3	17.8	21.6	35.8	23.2	**29.2**
CUG	52.4	52.0'	51.2	66.3	63.5*	40.6'	33.1'	33.5	54.5'	49.3'	**49.7**
SCDG	49.9'	55.1	52.5'	66.7	65.5*	44.2	35.0'	35.0	54.0'	52.2'	**51.0**
$\text{ECDG}^0_{0.0}$	31.7	49.2'	40.1'	56.9	40.4	40.7	27.3	18.5	11.1	49.6'	**36.5**
$\text{ECDG}^0_{0.2}$	34.7	55.4	46.3	57.1	46.2	41.3*	28.1	19.4	11.1	54.9'	**39.4**
$\text{ECDG}^0_{0.4}$	35.6	55.4	43.2	57.8	47.0	43.2	28.9	20.7	11.3	53.4	**39.6**
$\text{ECDG}^0_{0.6}$	33.6	51.6	36.9	57.4	47.2	40.9	28.7	21.1	11.4	47.8	**37.7**
$\text{ECDG}^0_{0.8}$	32.6	43.6	37.1	57.5	45.9	39.8	28.3	20.6	11.4	41.7	**35.9**
$\text{ECDG}^1_{0.0}$	29.0	50.5'	36.4	53.0	36.5	29.1	17.8	18.2	8.0	44.1	**32.3**
$\text{ECDG}^1_{0.2}$	30.7	53.7	37.8	53.0	40.0	29.3	18.2	17.5	8.0	46.1	**33.4**
$\text{ECDG}^1_{0.4}$	31.3	49.9	35.8	53.0	39.6	28.2	17.5	17.5	7.5	42.6	**32.3**
$\text{ECDG}^1_{0.6}$	29.6	44.5	30.7	52.8	37.5	26.5	17.1	18.3	6.9	36.4	**30.0**
$\text{ECDG}^1_{0.8}$	27.0	34.1	27.0	51.5	36.9	20.9	13.7	13.4	6.8	25.6	**25.7**
$\text{ECDG}^2_{0.0}$	27.3	40.6	31.1	47.1	32.9	13.8	10.6	11.0	6.4	31.8	**25.3**
$\text{ECDG}^2_{0.2}$	27.3	40.3	32.0	47.0	34.6	14.2	10.2	9.9	6.4	32.5	**25.4**
$\text{ECDG}^2_{0.4}$	28.0	37.0	31.0	46.5	32.8	12.9	9.9	11.0	5.8	30.2	**24.5**
$\text{ECDG}^2_{0.6}$	27.1	33.8	22.0	45.4	30.9	11.7	9.5	9.9	5.8	25.3	**22.2**
$\text{ECDG}^2_{0.8}$	23.3	25.2	17.5	44.6	30.0	6.8	8.6	8.8	6.0	16.7	**18.7**
Default_1	16.3	11.5	8.7	42.2	15.7	1.8	3.3	3.3	4.1	2.3	**10.9**
Default_m	11.3	1.2	3.2	2.1	0.4	2.8	5.9	4.4	2.1	10.7	**4.4**

more dependencies than the CIG and CAG (Table 3). For *CruiseControl (CC)*, *JHotDraw (JHD)*, and *Wicket (Wi)* the CIG even outperforms the CUG.

The SCDG, as an aggregation of the CIG, CAG, and CUG, incorporates the information from the three simple structural graphs and increases or at least stabilizes the clustering quality of the simple data sources: The average clustering quality of the SCDG (51.0) is clearly higher than the CIG and CAG average values (40.9 and 29.2) and at least slightly higher than the CUG value (49.7).

3.1.2 Results for Evolutionary Graphs

The $\text{ECDG}^0_{0.4}$, which has a support threshold of 0 and a confidence threshold of 0.4, produces the best average quality for evolutionary dependencies (39.6), closely followed by the other evolutionary graphs with a support threshold of 0 (please recall that a threshold of 0 means that the support must be at least 1). This clustering quality is in the range of the CIG (40.9), considerably better than the CAG (29.7), but clearly lower than the CUG (49.7) and SCDG (51.0) values.

These characteristics of the average value need not be valid for each of the individual projects. For instance, the evolutionary clustering quality for

the *Tomcat (Tom)* project is very low (5.8 to 11.4) and far from being competitive to any structural dependency information. This is probably caused by its sparse evolutionary class dependency graph (Table 3). In contrast, for *CruiseControl (CC)*, *JFreeChart (JFC)*, and *Wicket (Wi)* the relation between structural and evolutionary data sources is balanced—the best evolutionary and structural results are nearly equal.

3.1.3 Default Clustering Decomposition

Additional to the clustering results, Table 4 lists two default quality metric values in the last rows: Default_1 represents a decomposition that consists of only one huge cluster; Default_m represents a decomposition that consists of $m = |C(S)|$ singleton clusters (i.e., each cluster contains only one class). These values provide a reference for the *MoJoFM* values of a project.

The default decomposition qualities cover a wide range of *MoJoFM* values, from 0.4 (*JEdit (JE)*, Default_m) up to 42.2 (*iText (iT)*, Default_1). This observation underpins clearly that comparisons of clustering results based on *MoJoFM* are only valid for the same reference decomposition (i.e., the same sample software project). The *MoJoFM* difference between a clustering and the best default clustering gives a hint at the overall clustering quality. In contrast, taking the absolute value into account might be misleading. For instance, the structural clustering seems to work better for *JEdit (JE)* (SCDG: 65.5; Default_1: 15.7) and *Tomcat (Tom)* (CUG: 54.5; Default_1: 4.1) than for *iText (iT)* (SCDG: 66.7; Default_1: 42.2) although their best absolute *MoJoFM* values are similar. Regarding the evolutionary dependencies, only *Tomcat (Tom)* does not exceed its default values clearly, but at least slightly.

Result 1 *The usage graph, as well as the aggregated structural graph, outperforms all other data sources. The slightly filtered evolutionary dependencies produce results similar to the inheritance dependencies and better than aggregation dependencies. In nine of ten cases these evolutionary decompositions seem to be meaningful as their results exceed the default decompositions clearly.*

3.2 Experiment 2: Dependency Quality

Support and confidence are established metrics to measure the strength of evolutionary dependencies. This second experiment investigates whether stronger evolutionary dependencies are more reliable for software clustering. Furthermore, we want to examine the interplay between reliability and density of evolutionary dependencies.

We call the edges of a dependency graph that connect classes of the same package *intra-edges*. They enable the clustering algorithm to retrieve the package structure. In contrast, edges that connect classes from different packages influence the clustering result negatively. Hence, the percentage of intra-edges among all edges of the graph provides a measure of the dependency reliability

for software clustering. This measure is independent of the total amount of available dependencies. Table 5 lists the values of this intra-edge ratio (first value) for the previously used set of graphs. Additionally, the data density is expressed as the percentage of nodes with at least one in- or outgoing edge (second value), which we denote as *node coverage*.

Table 5 Percentage of package intra-edges (first value) and node coverage (second value)

(% values)	*Az*	*Che*	*CC*	*iT*	*JE*	*JFC*	*JHD*	*JU*	*Tom*	*Wi*	***avg***
CIG	25;60	39;84	37;72	73;59	70;44	69;50	59;61	31;54	47;51	53;88	**50;62**
CAG	42;52	60;32	72;45	77;56	81;64	41;29	39;48	51;37	46;61	44;50	**55;47**
CUG	32;98	39;99	32;96	65;98	59;99	27;97	23;85	33;84	35;94	26;99	**37;95**
SCDG	31;99	40;99	31;98	65;99	59;100	29;98	24;86	33;84	35;94	27;100	**38;96**
$ECDG^0_{0.0}$	16;54	37;95	32;87	77;34	23;65	40;66	20;63	14;48	30;15	24;94	**31;62**
$ECDG^0_{0.2}$	32;54	57;94	46;86	82;34	36;64	47;66	24;63	16;48	31;15	42;93	**41;62**
$ECDG^0_{0.4}$	39;51	76;89	51;83	83;33	43;64	54;64	27;63	16;48	35;14	50;90	**48;60**
$ECDG^0_{0.6}$	45;45	86;72	63;65	95;30	51;62	67;57	32;62	19;43	37;14	58;78	**55;53**
$ECDG^0_{0.8}$	47;39	84;60	65;52	95;28	53;60	63;56	30;61	16;40	37;14	56;65	**55;47**
$ECDG^1_{0.0}$	19;40	62;67	49;62	98;16	29;49	78;36	36;31	26;31	48;06	38;71	**48;41**
$ECDG^1_{0.2}$	36;39	77;66	66;61	98;16	43;49	83;36	41;31	27;30	46;06	56;71	**57;41**
$ECDG^1_{0.4}$	45;34	86;59	70;57	99;16	54;48	89;34	49;30	27;29	49;06	64;63	**63;38**
$ECDG^1_{0.6}$	45;30	90;49	69;44	98;15	65;45	94;32	52;29	24;28	56;04	70;53	**66;33**
$ECDG^1_{0.8}$	51;21	90;30	77;28	100;11	83;41	99;22	65;20	20;22	71;03	75;32	**73;23**
$ECDG^2_{0.0}$	20;32	72;42	59;43	100;07	33;39	85;16	47;14	24;13	67;03	45;50	**55;26**
$ECDG^2_{0.2}$	36;31	86;42	77;41	100;07	47;39	87;15	49;14	24;13	65;03	62;50	**63;26**
$ECDG^2_{0.4}$	44;27	89;38	76;37	100;07	59;38	87;14	49;13	29;13	61;02	69;44	**66;23**
$ECDG^2_{0.6}$	45;21	93;30	78;21	100;05	76;34	98;12	50;11	33;12	69;02	81;34	**72;18**
$ECDG^2_{0.8}$	49;13	98;17	87;10	100;04	90;31	100;05	52;09	20;09	82;02	91;18	**77;12**

3.2.1 Dependency Reliability

The average values in the last column of Table 5 show similar average intra-edge ratios for the structural graphs, ranging from 38% to 55%, and more varying ratios for evolutionary graphs, ranging from 27% to 77%. The lower reliability of totally unfiltered evolutionary dependencies explains their lower clustering quality in the previous experiment. With a stronger filter setup, the evolutionary dependencies, however, provide better dependency reliability. But this does not automatically imply better clustering results.

3.2.2 Dependency Density

Although we covered considerable time spans of development, the average node coverage rate of the evolutionary dependencies is 62% at most. In other

words, on average there is no evolutionary information available for 38% of the system. We had to ignore the initial check-in because it does not provide any reliable co-change information and the developers just did not change the files in the considered time span (see Table 1). In contrast, the CUG and SCDG nearly cover the whole system: their average node coverage rate is 95% (CUG) and 96% (SCDG)—the remaining nodes could be dead code, code loaded by reflections, or code only containing constants or meta-information.

The low coverage rates of the evolutionary dependencies are clearly the main problem of a clustering approach exclusively based on this kind of data: Many parts of the system just do not get changed over years. Assuming that classes that frequently changed in the past will also frequently change in the future, those are much more relevant in many software clustering applications. The evolutionary dependencies at least cover these potentially critical parts of the system. Nevertheless, structural dependencies obviously remain the first choice when complete coverage is important.

3.2.3 Filtering

The general trend with respect to the intra-edge percentages shows that the dependency reliability increases significantly at the costs of lower coverage rates. In contrast, varying the support only enhances the reliability moderately, but decreases the coverage rate faster. All in all, filtering by confidence seems to be more successful for the application of software clustering than filtering by support.

When looking at the single projects, we also observe outliers to these general tendencies: For instance, filtering evolutionary dependencies with a certain confidence value decreases the intra-edge ratio in the *JUnit (JU)* project. These counter-trend anomalies seem to be more frequent for stronger filtering setups, where the evolutionary information is already sparse.

Result 2 *The dependency density, not the dependency reliability, is the main problem of the evolutionary dependencies and explains their lower clustering quality. Nevertheless, it is important to filter the evolutionary data to exclude unreliable dependencies. Filtering by confidence works better than filtering by support.*

3.3 Experiment 3: Combined Data Sources

Clustering exclusively based on evolutionary data was only partly successful because the data density is often too low for a complete clustering. Since the data quality of the filtered evolutionary dependencies is good, yet sparse, in this experiment we integrated the evolutionary dependencies into the dense structural data to improve the overall clustering results.

For this experiment we only considered a subset of the previously used graphs because the expected extra gain would not justify the extra effort of

combining each of the four structural graphs with each of the 15 evolutionary graphs. SCDG, as the provider of the best structural clustering result, will be the representative of the structural graphs while the following selection of evolutionary graphs represents the evolutionary data: the raw data ($\text{ECDG}^0_{0.0}$), the best setup in the first experiment ($\text{ECDG}^0_{0.4}$), two trade-offs between reliability and coverage ($\text{ECDG}^0_{0.8}$ and $\text{ECDG}^1_{0.4}$), and a setup focusing on reliability ($\text{ECDG}^1_{0.8}$).

3.3.1 Simple Union

A straightforward method to integrate structural and evolutionary dependencies is a union operation on graphs.

Definition 5 Given two unweighted directed graphs, $G_1 = (V, E_1)$ and $G_2 = (V, E_2)$, the **graph union operation** $\cup$ applied to G_1 and G_2 creates an unweighted directed graph

$$G_1 \cup G_2 := G_3 = (V, E_3)$$

with $E_3 := E_1 \cup E_2$ (here, $\cup$ denotes the normal set union operation).

Table 6 presents the new clustering results for the combined data sources and contrasts them to the results of the first experiment. Please recall the usage of the symbols ' (standard error of $0.5 \leq \hat{\sigma}_{\overline{x}} < 1.0$) and * (standard error of $1.0 \leq \hat{\sigma}_{\overline{x}}$).

Table 6 MoJoFM clustering quality based on combined structural and evolutionary dependency graphs using the union operation

$n = 50,\ \hat{\sigma}_{\overline{x}} \leq 2.2$	*Az*	*Che*	*CC*	*iT*	*JE*	*JFC*	*JHD*	*JU*	*Tom*	*Wi*	***avg***
SCDG	49.9'	55.1	52.5'	66.7	65.5*	44.2	35.0'	35.0	54.0'	52.2'	**51.0**
$\text{ECDG}^0_{0.0}$	31.7	49.2'	40.1'	56.9	40.4	40.7	27.3	18.5	11.1	49.6'	**36.5**
$\text{ECDG}^0_{0.4}$	35.6	55.4	43.2	57.8	47.0	43.2	28.9	20.7	11.3	53.4	**39.6**
$\text{ECDG}^0_{0.8}$	32.6	43.6	37.1	57.5	45.9	39.8	28.3	20.6	11.4	41.7	**35.9**
$\text{ECDG}^1_{0.4}$	31.3	49.9	35.8	53.0	39.6	28.2	17.5	17.5	7.5	42.6	**32.3**
$\text{ECDG}^1_{0.8}$	27.0	34.1	27.0	51.5	36.9	20.9	13.7	13.4	6.8	25.6	**25.7**
$\text{SCDG}\cup\text{ECDG}^0_{0.0}$	48.4'	51.0	46.4	65.0'	54.8*	53.1	36.1'	28.2	51.7'	50.9'	**48.6**
$\text{SCDG}\cup\text{ECDG}^0_{0.4}$	50.6'	59.8	52.7	65.7	63.7*	55.9	39.7	31.9	56.1	58.4'	**53.4**
$\text{SCDG}\cup\text{ECDG}^0_{0.8}$	52.7	60.4'	54.2	67.8	68.7'	53.3*	37.4'	34.4	55.9	56.9'	**54.2**
$\text{SCDG}\cup\text{ECDG}^1_{0.4}$	52.3	63.0	50.6'	68.9	64.6*	56.0	38.1	32.7	57.0	57.8'	**54.1**
$\text{SCDG}\cup\text{ECDG}^1_{0.8}$	52.6	59.7	54.3	68.1	64.1*	50.5'	37.3	35.1	56.6	55.4	**53.4**

Combining structural and filtered evolutionary dependencies by using the simple union operation improves the average clustering quality from 51.0 (SCDG) to 54.2 ($\text{SCDG}\cup\text{ECDG}^0_{0.8}$). It can be rejected with sufficient confidence that this could be a random effect: We compared the results of the

SCDG and the results of all combined graphs including filtered evolutionary information. The latter seem to perform better than the SCDG; a Friedman Test—a non-parametric statistical test for independent measures—rated the difference as statistically significant ($p = 0.014$; significance level: 5%).

In contrast to Experiment 1, a stronger filtering of evolutionary dependencies provides the best results (Experiment 1: $\text{ECDG}^0_{0.4}$; current experiment: $\text{SCDG} \cup \text{ECDG}^0_{0.8}$). When combining data sources, the reliability of evolutionary dependencies seems to be more important than their coverage. Integrating unfiltered evolutionary dependencies even decreases the clustering quality from 51.0 to 48.6 for the ten analyzed projects.

Result 3 *The quality of a clustering based on structural dependencies increases when integrating filtered evolutionary dependencies.*

3.3.2 Weighted Union

Both structural as well as evolutionary dependency information may be flawed: A particular structural dependency may exist because a developer has misplaced a method. Similarly, a particular evolutionary dependency may exist because two classes were changed coincidentally at the same time. But if both dependencies link the same two classes, it is unlikely that this happens just by chance. An analysis of the dependency quality in terms of intra-edge ratios (compare to Experiment 2) shows the importance of this effect.

Table 7 presents the intra-edge and node coverage results for duplicate dependencies—i.e., dependencies included in the intersection of both original graphs, which we define analogously to the union operation: $(V, E_1) \cap (V, E_2) := (V, E_1 \cap E_2)$. The intra-edge ratio of the duplicate dependencies (49% to 74%) is clearly higher than the intra-edge ratio of the original dependencies (structural: 38%; evolutionary: 31% to 73%). Moreover, the intersection still covers 8% to 42% of the nodes. If those nodes can be clustered better, this might make a difference for the clustering result. Thus, we try to use this increased reliability to improve the clustering results.

After integrating both data sources with the union operation, the relevance of all dependencies is identical. To exploit the more reliable duplicate dependencies, we introduce a dependency importance implemented as an edge weight. An extended union operation on graphs allows assigning a higher edge weight to duplicate dependencies. We will denote the resulting three sets of dependencies as *dependency groups* in the following.

Definition 6 Given two unweighted directed graphs, $G_1 = (V, E_1)$ and $G_2 = (V, E_2)$, the **weighted graph union operation** $\cup_{[a,b,c]}$ with $a, b, c \in \mathbb{R}$ applied to G_1 and G_2 creates a weighted directed graph

$$G_1 \cup_{[a,b,c]} G_2 := (V, E_3, \mu)$$

Table 7 Percentage of package intra-edges (first value) and node coverage (second value) for duplicate dependencies

(% values)	*Az*	*Che*	*CC*	*iT*	*JE*	*JFC*	*JHD*	*JU*	*Tom*	*Wi*	***avg***
SCDG	31;99	40;99	31;98	65;99	59;100	29;98	24;86	33;84	35;94	27;100	**38;96**
$ECDG^{0}_{0.0}$	16;54	37;95	32;87	77;34	23;65	40;66	20;63	14;48	30;15	24;94	**31;62**
$ECDG^{0}_{0.4}$	39;51	76;89	51;83	83;33	43;64	54;64	27;63	16;48	35;14	50;90	**48;60**
$ECDG^{0}_{0.8}$	47;39	84;60	65;52	95;28	53;60	63;56	30;61	16;40	37;14	56;65	**55;47**
$ECDG^{1}_{0.4}$	45;34	86;59	70;57	99;16	54;48	89;34	49;30	27;29	49;06	64;63	**63;38**
$ECDG^{1}_{0.8}$	51;21	90;30	77;28	100;11	83;41	99;22	65;20	20;22	71;03	75;32	**73;23**
$SCDG \cap ECDG^{0}_{0.0}$	37;50	60;83	39;82	82;30	55;63	59;45	36;50	42;44	45;11	38;90	**49;55**
$SCDG \cap ECDG^{0}_{0.4}$	60;29	78;41	56;41	86;24	72;49	58;27	50;34	38;33	42;08	62;55	**60;34**
$SCDG \cap ECDG^{0}_{0.8}$	63;15	81;18	71;18	94;18	87;40	56;11	55;19	46;17	40;05	76;24	**67;18**
$SCDG \cap ECDG^{1}_{0.4}$	61;21	82;27	60;26	100;11	74;37	70;13	54;11	37;19	58;03	56;38	**65;21**
$SCDG \cap ECDG^{1}_{0.8}$	58;09	84;07	65;10	100;07	93;28	75;03	73;03	20;06	100;01	68;11	**74;08**

where $(V, E_3) = G_1 \cup G_2$ and $\mu : E_3 \to \mathbb{R}$ is a weight function defined as

$$\mu(e) := \begin{cases} a & \text{if } e \in E_1 \wedge e \notin E_2 \\ b & \text{if } e \in E_1 \wedge e \in E_2 \\ c & \text{if } e \notin E_1 \wedge e \in E_2 \end{cases}$$

Note that this definition allows weights of 0, which will result in ignoring the corresponding dependencies in the clustering process. A weighted union with weights of 1 for all three parameters is equivalent to the simple union.

These weights influence the *Bunch* clustering tool, or more exactly, its internal quality metric. To emphasize the importance of the duplicate dependencies, their weight should be higher than that of the non-duplicate dependencies. To make a clear difference, we set the weight of the duplicate ones to 4 while the other edge weights stay 1. This setup is reflected in the weighted union operation $\cup_{[1,4,1]}$. Table 8 compares the clustering results based on this weighted union operation to the previous results.

The clustering qualities of the weighted combinations enhance slightly in comparison to the non-weighted combinations. This effect is statistically significant as a Wilcoxon Test[4] that compares the project average of non-weighted union clustering results to the equivalent average of weighted union results shows ($p = 0.022$; significance level: 5%). The best clustering result increases from 54.4 to 55.3. The largest differences can be, however, observed for the non-filtered evolutionary dependencies (from 48.6 to 52.9). In general, using the weighted union enlarges the difference between the best combined strategy and the exclusively structural clustering (SCDG: 51.0; $SCDG \cup ECDG^{0}_{0.8}$: 55.3).

[4] The Wilcoxon Test is used instead of a Friedman Test when only comparing two variables instead of three or more.

Table 8 MoJoFM clustering quality based on combined structural and evolutionary dependency graphs using the weighted union operation $\cup_{[1,4,1]}$

$n = 50,\ \hat{\sigma}_{\overline{x}} \leq 2.2$	*Az*	*Che*	*CC*	*iT*	*JE*	*JFC*	*JHD*	*JU*	*Tom*	*Wi*	***avg***
SCDG	49.9’	55.1	52.5’	66.7	65.5*	44.2	35.0’	35.0	54.0’	52.2’	**51.0**
SCDG∪ECDG$^0_{0.0}$	48.4’	51.0	46.4	65.0’	54.8*	53.1	36.1’	28.2	51.7’	50.9’	**48.6**
SCDG∪ECDG$^0_{0.4}$	50.6’	59.8	52.7	65.7	63.7*	55.9	39.7	31.9	56.1	58.4’	**53.4**
SCDG∪ECDG$^0_{0.8}$	52.7	60.4’	54.2	67.8	68.7’	53.3*	37.4’	34.4	55.9	56.9’	**54.2**
SCDG∪ECDG$^1_{0.4}$	52.3	63.0	50.6’	68.9	64.6*	56.0	38.1	32.7	57.0	57.8’	**54.1**
SCDG∪ECDG$^1_{0.8}$	52.6	59.7	54.3	68.1	64.1*	50.5’	37.3	35.1	56.6	55.4	**53.4**
SCDG$\cup_{[1,4,1]}$ECDG$^0_{0.0}$	53.6’	58.4	49.9	65.5’	64.7*	52.1*	39.1	35.1	52.6’	57.4	**52.9**
SCDG$\cup_{[1,4,1]}$ECDG$^0_{0.4}$	52.8’	63.0	53.4	67.9	67.9*	54.8’	39.7	33.2	55.1’	60.5	**54.8**
SCDG$\cup_{[1,4,1]}$ECDG$^0_{0.8}$	52.7’	60.5	55.4	69.0’	71.1’	54.4’	38.4	37.1	55.4	58.8	**55.3**
SCDG$\cup_{[1,4,1]}$ECDG$^1_{0.4}$	53.1’	63.2	53.7	67.1’	69.0*	54.8’	37.8	34.8	55.3’	59.4	**54.8**
SCDG$\cup_{[1,4,1]}$ECDG$^1_{0.8}$	52.8	59.6	53.4*	67.0’	70.4’	50.0’	37.2	34.9	56.2	56.9	**53.9**

Result 4 *The reliability of duplicate dependencies is better than the reliability of general structural or evolutionary dependencies. The clustering results show that emphasizing these duplicate dependencies in the integration of structural and evolutionary data improves the clustering slightly further.*

3.3.3 Parameter Optimization

The weights in the previous experiment were derived from theoretical considerations. Nevertheless, better weighting setups may exist, which should be found by systematically varying the weights in a reasonable range. The following sub-experiment implements such a weight optimization by comparing the clustering qualities of differently weighted combined graphs.

To get results in due time, we decided to vary the weights in five steps, resulting in $5^3 = 125$ different weight setups. While the SCDG stands for the structural data, the ECDG$^0_{0.8}$, which has produced the best results in combination with the SCDG up to now, represents the evolutionary data. The search space is covered by the set of weights $\{0, 1, 2, 4, 8\}$ for each weight parameter.

Table 9 documents the parameter optimization experiment by a selection of the best twelve clustering results with respect to the average *MoJoFM* similarity.

The first and most important conclusion from the results is that the clustering quality does not improve in comparison to the weighted union $\cup_{[1,4,1]}$ ($\cup_{[2,8,2]}$ is equivalent and produces similar results—slight variations are due to the random aspect in *Bunch*). This confirms our preliminary considerations on emphasizing the duplicate dependencies. Nevertheless, one should not overstate their importance: $\cup_{[1,8,1]}$ is already rated lower.

Additionally, we analyzed the entire list of 125 setups and observed some further trends:

Table 9 Best MoJoFM clustering qualities based on the combined SCDG and $\text{ECDG}^0_{0.8}$ using the weighted union operation in different setups

$n = 50,\ \hat{\sigma}_{\overline{x}} \leq 1.8$	*Az*	*Che*	*CC*	*iT*	*JE*	*JFC*	*JHD*	*JU*	*Tom*	*Wi*	***avg***
$\text{SCDG} \cup_{[1,4,1]} \text{ECDG}^0_{0.8}$	52.7'	60.5	55.4	69.0'	71.1'	54.4'	38.4	37.1	55.4	58.8	**55.3**
$\text{SCDG} \cup_{[2,8,2]} \text{ECDG}^0_{0.8}$	53.2	60.2	55.0'	69.0'	71.4'	54.0'	38.7	37.3	55.9	58.2	**55.3**
$\text{SCDG} \cup_{[1,8,1]} \text{ECDG}^0_{0.8}$	51.2*	60.1'	56.8	68.5	70.7*	53.4*	38.0	36.4	56.4	57.3	**54.9**
$\text{SCDG} \cup_{[1,2,1]} \text{ECDG}^0_{0.8}$	52.6	61.2	55.3'	70.2	69.3*	53.1*	38.6	35.3	56.1	56.6*	**54.8**
$\text{SCDG} \cup_{[2,4,2]} \text{ECDG}^0_{0.8}$	52.3'	60.0'	54.9	68.1'	70.0'	54.9	38.2	35.2	55.6	58.5	**54.8**
$\text{SCDG} \cup_{[4,8,4]} \text{ECDG}^0_{0.8}$	53.1	60.7	54.8'	69.8	70.7	53.8'	38.2	34.9	54.8'	57.3'	**54.8**
$\text{SCDG} \cup_{[1,8,2]} \text{ECDG}^0_{0.8}$	50.9'	58.2'	56.2	67.7	69.3*	55.4'	38.0	36.5	56.9	58.1	**54.7**
$\text{SCDG} \cup_{[2,2,2]} \text{ECDG}^0_{0.8}$	52.2	61.2	53.6	67.5	69.4	54.3	38.0	34.3	56.2	58.6	**54.5**
$\text{SCDG} \cup_{[2,8,1]} \text{ECDG}^0_{0.8}$	53.4	58.4	55.7'	69.2'	69.2*	51.1'	38.3	35.9	55.4	56.9	**54.4**
$\text{SCDG} \cup_{[4,4,2]} \text{ECDG}^0_{0.8}$	53.0	60.1	55.1	69.0	69.7'	52.1	38.6	36.3	54.9	55.7'	**54.4**
$\text{SCDG} \cup_{[4,8,2]} \text{ECDG}^0_{0.8}$	52.6'	59.5	56.0	69.4'	70.8'	51.1'	38.1	34.7	55.4	56.0'	**54.4**
$\text{SCDG} \cup_{[8,8,8]} \text{ECDG}^0_{0.8}$	51.8'	61.5	53.5	66.8'	69.6	54.5	38.2	34.1	55.4	58.3	**54.4**
				...							

- Ignoring one of the dependency groups does not produce good clustering results. Setups that consider all groups (no weight is 0) reach an average quality of 53.7. Leaving out exclusively structural dependencies (only the first weight is 0) produces the worst results of 35.6. Since the other two groups are much smaller, omitting exclusively evolutionary or duplicate dependencies is not as dramatic (exclusively evolutionary: 51.4; duplicate: 53.2).
- Comparing all combinations that consider all dependency groups, setups where the weight of the duplicate dependencies is among the highest perform slightly better (54.1) than setups where it is among the lowest weights (53.7).
- The importance of the structural-only dependencies is about equal to the importance of the evolutionary-only dependencies. An average over all setups shows that setups with a higher structural than evolutionary weight produce an average quality of 53.5 while setups with a higher evolutionary than structural weight produce a quality of 53.2. But as we observed in further experiments, a different filtering setup could change this relation.

Result 5 *Each of the dependency groups is valuable. The parameter optimization underlines again that duplicate dependencies are most important. The best weight distribution of exclusively structural and evolutionary dependencies depends on the strength of the evolutionary filtering.*

3.4 Detailed Analysis

The results presented so far draw general conclusions by summarizing the quality of the clustering decompositions on a high level of abstraction. In

this section we deepen our understanding of these results by picking up interesting parts of the results and finding reasons that explain the particular phenomenon.

The *MoJoFM* metric provides a measure to estimate the quality of a clustering decomposition. Nevertheless, it is hard to judge the practical applicability of a clustering decomposition by just looking at its *MoJoFM* value. Hence, our first step will be to relate the metric values to a more intuitive representation in a small example.

When comparing the clustering decomposition to the package structure, it would be interesting to know which package is matched by which cluster. Thus, we identify the most similar cluster among all clusters of the clustering decomposition for each package. An appropriate metric to estimate the similarity of a package and a cluster—two sets of classes—is the Jaccard coefficient. It measures the size of the overlap in relation to the total size of the package and the cluster. When the package represents a set of classes A and the cluster a set of classes B, the Jaccard similarity coefficient can be formally defined as

$$\mathrm{sim}(A, B) := \frac{|A \cap B|}{|A \cup B|}.$$

A similarity value of 0.5, for instance, means that the classes contained in both the package and the respective cluster cover 50% of the union of package and cluster. In the following we denote package–cluster pairs as *well-matched* for a Jaccard similarity of more than 0.75, *reasonably-matched* of more than 0.5, *partly-matched* of more than 0.25, and *non-matched* for worse similarity values.

Table 10 shows an example where we compared the main packages of the *Azureus* project to three different clustering decompositions—a structural-based one (SCDG), an evolutionary-based one ($\mathrm{ECDG}^0_{0.4}$), and a combined one ($\mathrm{SCDG} \cup_{[1,4,1]} \mathrm{ECDG}^0_{0.8}$). Since we have to use a concrete clustering decomposition here, we cannot summarize multiple runs of the clustering algorithms as done before. Instead, we just take a random example out of all created clustering decompositions. For each of the main packages, we provide the Jaccard similarity values to the two most similar clusters in the clustering decomposition. For choosing the second cluster, we ensured that it is neither a subset nor a superset of the first cluster.

In Table 10, the Jaccard similarity values above 0.25 (partly-matched) are highlighted in light gray and above 0.5 (reasonably-matched) in gray. The second similarity value provides a clue how the rest of the package not covered by the best-matching cluster is matched—two values, both higher than 0.25, also indicate a sound clustering result. For instance, in the clustering decomposition based on the structural information (Table 10, left), three packages are reasonably-matched and two further ones partly. While the combined results are similar (Table 10, right), the evolutionary dependency results cannot compete with these results (Table 10, middle). These observations match the

Table 10 Best-matching clusters for the main packages of the Azureus project

Package	SCDG		$ECDG^0_{0.4}$		$SCDG \cup_{[1,4,1]} ECDG^0_{0.8}$	
	1st	2nd	1st	2nd	1st	2nd
`disk`	0.68	0.07	0.24	0.10	0.73	0.15
`download`	0.52	0.09	0.26	0.09	0.44	0.23
`peer`	0.43	0.19	0.29	0.13	0.32	0.26
`torrent`	0.39	0.35	0.22	0.15	0.46	0.35
`tracker`	0.53	0.19	0.43	0.13	0.50	0.27
`util`	0.23	0.19	0.21	0.18	0.25	0.20

results based on the *MoJoFM* metric for the *Azureus* project (Table 4 and Table 8).

The packages with a higher similarity coefficient for structural information are probably cohesive because the clustering algorithm is able to retrieve them to some extent. But still they could not be perfectly confirmed by the algorithm. Since the top similarity values do not increase, integrating evolutionary information does not clearly change the situation. Hence, we took the `disk` package, which provides the best results for structural and combined information, and investigated why some of its classes could not be added to the correct cluster. Our observation is that the rest of the `disk` package builds a small cluster on its own (the second similarity value in the columns for combined information of Table 10). This part of the package is somewhat more related to the `download` package than to the rest of the `disk` package. Probably, the designers of the system could have as well assigned the respective classes to the `download` package. The algorithm cannot be expected to solve such an ambiguous situation.

Beyond analyzing a single project, the approach of finding best-matching clusters for important packages can be extended to covering all projects. We take into account all packages that at least contain 10 classes but do not cover more than 50% of the project. Those are compared to one clustering decomposition per project and data source. Figure 1 summarizes the retrieved Jaccard similarity values for the best-matching cluster for each of the packages. Each of the three depicted histograms represents one data source.

The histograms in Figure 1 confirm the results derived from the *MoJoFM* values (Table 4 and Table 8): Structural as well as combined structural and evolutionary dependencies provide clustering results that match the package structure best. The clusters produced by only employing evolutionary information cannot compete as the lower similarity values suggest. Comparing the first and the third histogram, the main difference seems to consist of the much lower first bar and the much larger second bar in the third histogram. That means, combining the data sources mainly improves the retrieval of the previously non-matched packages but only slightly improves the results of the already partly- and reasonably-matched packages. This observation provides a more comprehensive illustration of what a MoJoFM improvement from 51.0

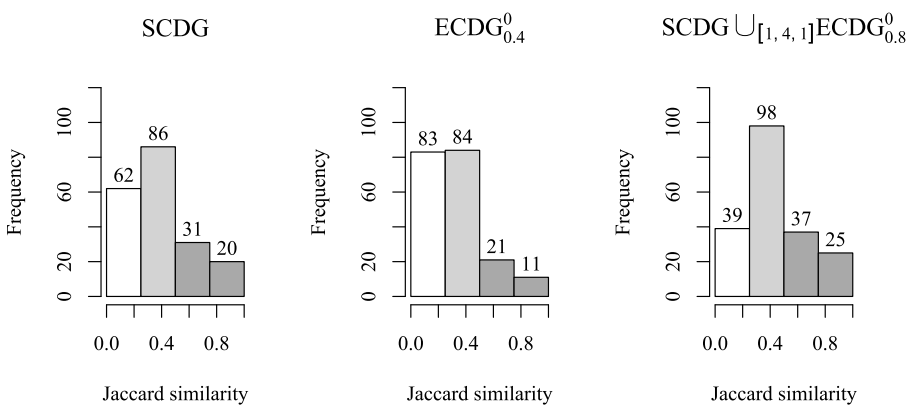

Fig. 1 Jaccard similarity distribution for the best-matching clusters of the packages of all analyzed software projects

(SCDG) to 55.3 (SCDG$\cup_{[1,4,1]}$ECDG$^0_{0.8}$) can mean. Clustering applications where it is important to find a reasonable cluster for every class could considerably profit from the combined data sources.

Finally, we want to find reasons why some of the packages do not have a matching counterpart in the clustering hierarchy. For the *Azureus* project, the `util` package embodies such an example (Table 10). In the histograms that summarize all projects, we look at the non-matched packages. (Figure 1, first bar). The results of the combined dependency graphs still include 39 such packages (Figure 1, right). We found explanations for 27 of the 39 packages why they could not be matched:

- *Eight* packages are utility packages, which are hard to retrieve by a dependency-based software clustering algorithm because of their inherent structure: The joint characteristic of the classes in such a package usually is that they are accessed from all over the project without having dependencies to classes outside the utility package—they are not highly cohesive themselves. Such omnipresent clusters like utility packages, however, can be detected in a preprocessing step (Wen and Tzerpos, 2005). Further *eight* non-matched packages of *JHotDraw* have a similar non-cohesive structure because they aggregate sample code.
- The granularity of the hierarchical clustering decomposition does not always match the granularity of the package structure. The absence of a matching level of granularity explains why further *eight* of the 39 packages could not be appropriately retrieved. Many of the affected packages belong to the *JEdit* project, where the clustering algorithm already splits the system into 108 clusters on the first level of the hierarchical decomposition. Hence, the clusters are much too fine-grained for matching one of the larger packages of the system. Forcing the algorithm to add another, more coarse-grained layer of clusters would probably solve this issue.

– Further *three* non-matched packages are part of the *JFreeChart* project. In this project, test classes are mixed with normal classes. The clustering algorithm tends to cluster these test classes together. In the package structure, the developers, however, assigned test classes to the classes that are tested. Hence, the three non-matched packages could be largely blamed to these two different paradigms to handle test classes.

These analyses showed that the intuitive and detailed comparison of packages and clusters confirmed the general results proposed by the *MoJoFM* values: The results are much better for structural and combined data sources than for evolutionary dependencies. It is interesting to observe that the combination does not uniformly improve the results, but mainly reduces the number of non-matched packages, which can be important in many practical applications. Several of the still non-matched clusters can be explained by repairable deficiencies of the clustering algorithm.

3.5 Summary

In the three clustering experiments and the subsequent detailed analysis we observed that software clustering can produce meaningful results based on structural as well as on evolutionary data: The results clearly exceed the default decompositions in nearly all cases. The overall quality of the clustering results based on these individual data sources is, however, much better for two of the structural dependency types (usage and aggregated structural dependencies). Analyses of the dependency quality and density showed that the lower clustering quality of the evolutionary dependencies can probably be attributed to their sparse node coverage (62% for unfiltered dependencies; lower for filtered dependencies).

Combining structural and evolutionary dependencies improves the clustering results—even slightly more when giving dependencies that mutually appear in both graphs a higher weight. While these improvements are statistically significant, the detailed analysis illustrated the effect of the improvement: Many previously non-matched packages are at least partly matched in the clustering results of the combined data sources. Depending on the particular software clustering application, this could be an important advantage.

4 Threats to Validity

This section discusses the validity of our empirical approach. Following Cook and Campbell (1979) we distinguish *conclusion validity*, *internal validity*, *construct validity*, and *external validity*.

Conclusion Validity—*Is there a relationship between the varied input and the observed outcome?*

We systematically varied the input by testing different dependency graphs. Differences in the results can be either attributed to these changes of the input or to random effects. We applied inference statistics and could verify the statistical significance of the result wherever a general result was not certain. Nevertheless, we also report observations specific to single projects or to a particular aspect of the input that did not undergo such a thorough investigation. These observations have to be considered only as preliminary findings.

Internal Validity—*Is the observed relationship causal?*

Our approach to measure the impact of different data sources on the application of software clustering is indirect and employs several heuristics. Although, our goal was to design a fair study, which does not bias the result in any direction, we cannot guarantee perfect fairness. Possibly, each step in the experimental design could bias the result, for instance:

- The clustering algorithm might work better on dependency graphs having certain structures. Studying the evaluation of *Bunch* (Mitchell and Mancoridis, 2007), we did not, however, find any hints pointing in this direction.
- Using existing package structures as a reference comes along with some problems: It assumes that there exists a single optimal solution to the clustering problem. It furthermore takes only the structure into account as it is, and not as it should be. Nevertheless, we still believe this external assessment approach to be the most appropriate evaluation method available for our scenario (Section 2.3) and also tried to enhance the approach by selecting only those software projects that show certain quality attributes (Section 2.4).
- The clustering assessment only considers parts of the data: The approach is based on flat decompositions and hence ignores the hierarchical structure of the packages as well as of the clustering decomposition. By selecting the level in the clustering decomposition that matches best the lowest level of the package structure, we target at preserving the most significant information; but the result only reflects the probably best part of the clustering decomposition.

Construct Validity—*Do we measure the constructs as intended?*

The dependency graphs as defined in Section 2.1 may not model the theoretical construct of structural and evolutionary dependencies as intended. At least we used definitions similar to those used in literature. Furthermore, from a pragmatic point of view, everything that improves the clustering result is of interest.

Comparing two hierarchies and computing their similarity is far from being trivial. Although relatively intuitive and often applied, *MoJoFM* may not measure as expected the similarity of the clustering result to the reference decomposition.

External Validity—*Are the results generalizable?*

By default, the results of a study are only valid for the examined subjects—here, for the sample software projects in a certain version. The current study is

based on ten different software projects that cover a wide range of application types, however restricted to *Java* projects of less than 1000 classes. Nevertheless, the study is one of the most extensive studies in software clustering—only a few studies examine more software projects (Mitchell and Mancoridis, 2001, 2007) or larger projects (Beyer and Noack, 2005; Wu et al, 2005). Due to the wide range of studied software projects, the findings are generalizable to some extent and might be a good indication for other *Java* projects and at least a weak indication for general software projects.

Improving the clustering results of *Bunch* does not directly imply that this is also possible for other clustering approaches in the same way. It, however, is an indicator of an increased data quality for software clustering integrating structural and evolutionary data sources. Possibly, other approaches might also be able to use this potential improvement of data quality to produce better clustering results—replicating the experiment using other clustering techniques would be necessary to confirm this assumption.

Similarly, the use case of architecture recovery limits the validity of the results. Nevertheless, it is plausible that other use cases of software clustering would also profit from the applied approach of integrating structural and evolutionary dependencies. Especially, applications where completeness is not a prime requirement are predestined for the usage of evolutionary dependencies.

Moreover, the study only showed that it is possible to improve the clustering results by a certain setup, but it cannot make any statements about to what degree the potential of the data sources is already used. Although we aimed at using a high quality clustering setup, it may be possible to get much better results in a different setup (e.g., with a different data integration method or other clustering parameters).

5 Consolidation of Results

Recently, several research groups came up with approaches that compare and try to exploit different types of code dependencies similar to the approach presented in this paper. Since code dependencies are used in diverse software engineering applications, these approaches follow different goals but are based on similar ideas and observations. In this section we want to consolidate our results and the results of these related approaches. A brief survey may reveal additional support for particular results or hint at controversial findings.

5.1 Comparing Structural and Evolutionary Dependencies

The relevance of structural and evolutionary dependencies in software design can be traced back to two basic design paradigms for modular software: *information hiding* (Parnas, 1972) and the principle of *low coupling and high cohesion* (Stevens et al, 1974). Information hiding advocates hiding design decisions into modules—co-changes occurring mainly within module borders

would be the consequence. In contrast, low coupling between the different modules and high cohesion within a module address the structural dependencies among the elements of the modules.

Software Clustering: As already mentioned, structural and evolutionary information have been already collectively applied in software clustering (Andritsos and Tzerpos, 2005; Sindhgatta and Pooloth, 2007; Wierda et al, 2006). Since these approaches also introduce new software clustering techniques, the comparison of the two data sources is only a by-product. In contrast, our work consequently focuses on investigating the relation between the two coupling types in this application scenario. But especially the approach by Sindhgatta and Pooloth (2007) is closely related to ours as it is also based on co-change data derived from the transactions of the versioning system. Instead of transforming transactions into dependencies, they decided to transform the structural dependencies into transactions. Although they do not directly contrast structural and evolutionary information, a combination of both data sources improved the results of the formerly structural-based approach similar to our results.

Mining Concerns: Related to software clustering, some approaches in mining crosscutting and modular concerns include structural and evolutionary information. Breu and Zimmermann (2006) present a technique that is based on concurrently added method calls in the same transaction—structural and evolutionary information are thereby inseparably connected. Adams et al (2010) contrast this approach to an approach based on structural information only. Moreover, they modify and extend the evolutionary mining strategy in a third approach. The results of a case study show only small overlap between the three techniques; it seems that each technique covers a different dimension of concern mining, which could also be related to some extent to the different data sources they employ.

Change Impact Analysis: For predicting the impact of changes of source code, Hattori et al (2008) use evolutionary information to sort the results of a prediction based on structural information, which they demonstrate in a small case study. The approach by Zhou et al (2008) combines structural information and evolutionary information like frequency, significance, age, and author of a change and thereby outperforms an approach exclusively based on evolutionary information. Wong and Cai (2009) introduce another hybrid change prediction approach. In a case study, they compare the hybrid approach to an approach based on structural information and to one using only evolutionary information. While the structural approach performs well at the early phase of development, the evolutionary provides better results later in the evolution of the studied software project. In contrast, the hybrid approach yields competitive results over the whole span of development. We can learn from this study that it could be questionable to rely on evolutionary information at the early phase of development when evolutionary information is only sparsely available. Although we did not test different versions of the same project in our study, we also observe that the sometimes sparse data density of evolutionary dependencies is a major drawback of this data source.

Bug Prediction: Cataldo et al (2009) compare the impact of different code dependencies on failure proneness, among them structural and evolutionary dependencies, but also work dependencies, which model workflow dependencies and coordination requirements. Their results provide evidence that a high number of dependencies more reliably predicts bugs in case of evolutionary and work dependencies than in case of structural dependencies.

Summarizing these studies we observe that structural and evolutionary information seems to cover indeed different dimensions of code dependence. An explanation might be that many co-changes do not seem to relate to any change of the structure of the code, as Fluri et al (2005) report in a case study. But in general, it is yet unknown what the specific differences between the two data sources are.

Combining structural and evolutionary information could be beneficial for many applications, not only in software clustering. In the case of change impact analysis and software clustering, structural information seems to play the leading part while evolutionary dependencies only provide an extra. In bug prediction, there exist first hints that evolutionary dependencies are more important.

5.2 Other Types of Dependency

Not only structural and evolutionary data sources are integrated in software engineering applications. For instance, in software clustering applications, structural information provides the foundation to integrate other data sources like semantic coupling derived from identifier names and comments (Bittencourt et al, 2010; Bavota et al, 2010) or dynamic code dependencies observed during the execution of the software (Patel et al, 2009). But also co-change information has been the starting point to integrate other dependency information: Merging semantic and evolutionary coupling has been applied in change impact analysis (Kagdi et al, 2010). Evolutionary and code clone dependencies seem to complement each other when mining crosscutting concerns (Canfora et al, 2006).

These studies show that various types of code dependencies can be successfully combined in different applications. Moreover, Beck and Diehl (2011) analyzed the congruence of code dependencies and modularity. They found that not only different forms of structural dependencies and evolutionary dependencies seem to play an important role, but also semantic similarity and shared code-clones preferably connect classes of the same packages. Based on these observations we conjecture that more consequently integrating these dependency types will further improve the application at hand, for instance, the results of a software clustering approach.

5.3 Combination Strategies

When combining different dependency types, for instance, structural and evolutionary dependencies, the choice of the combination strategy might be one of the most crucial design decisions: We already observed differences when only switching our combination strategy from *union* to *weighted union*. Various strategies have been proposed—nearly every approach mentioned above uses a different strategy. We will give an overview on these strategies and categorize them in the following.

Merge the Results: A very simple strategy is to independently run the algorithm on the two data sources and only merge the two results. For instance, Kagdi and Maletic (2007) propose to unite or intersect the result sets in change impact analysis. In a similar scenario, Wong and Cai (2009) multiply the weights of two different predictions to merge the results. The main advantage of these approaches is that we are able to optimize the algorithm for each data source independently or even apply two totally different approaches. The results, however, must have a simple structure so that a combination is possible. Hence, applying this approach to software clustering is difficult because the result set is quite complex: If hierarchical clustering strategies are applied, we would have to merge two hierarchies.

Concatenate the Analyses: After performing a normal analysis on a single data source, we may use the result as the input for a second analysis employing a different data source. Bittencourt et al (2010) as well as Patel et al (2009) apply a second analysis in their clustering approach to those classes that could not be assigned to a package in the first step. Similarly, the second analysis can be used to filter or rank the results of a first analysis based on structural information by applying evolutionary information (German et al, 2009; Hattori et al, 2008). These kinds of concatenation strategies lead to a asymmetric combination of the two data sources; usually the data source of the first analysis has the higher impact on the result. Hence, it is reasonable to apply the more reliable or more conservative approach first, which often seems to be the approach based on structural information.

Merge the Dependency Graphs: The approach we followed in the current study was to merge the dependency graphs before performing the clustering. We applied set operations like union, weighted union, and intersection on the set of edges. Wierda et al (2006) also merge dependency graphs based on union and intersection, however, structural dependency graphs from different versions instead of dependency graphs from different data sources. Bavota et al (2010) combine two similarity matrices, which is equivalent to uniting two weighted complete graphs and averaging the edge weights. The change prediction approach by Zhou et al (2008) involves a more elaborate dependency merging strategy: They use different features of a relation to predict a summarized information; in other words, they train a prediction algorithm to combine n dependency types into one.

Merge the Underlying Information: Finally, it is also possible to merge the information from different data sources that relate code entities to each

other without actually coming up with a notion of code dependencies. For instance, the transaction-based clustering approach by Sindhgatta and Pooloth (2007) transforms the structural dependency information into transactions, which can be used equivalently to co-change transactions. Another approach is to describe each software entity as a feature vector whereas the features may stem from totally different data sources (Andritsos and Tzerpos, 2005). The hybrid concern mining approaches combine structural and evolutionary information by interpreting the evolutionary changes in the structural dependency graph and use these changes as seeds for identifying concerns (Breu and Zimmermann, 2006; Adams et al, 2010).

The four categories of merging strategies show that it is possible to combine the information from several data sources at different stages of the analysis. While the strategy of merging the underlying information or the graph structure is conducted before running an analysis algorithm (e.g., a clustering algorithm), the concatenation of analyses and the merging of results is applied after at least one analysis. In consequence, the pre-analysis strategies rely on more symmetric approaches because both data sources are handled equally in the analysis process; the post-analysis merging strategies allow totally asymmetric solutions. There exist different ways to implement a symmetric combination strategy, which can be grouped based on the data structure they employ: feature vectors, transactions, or dependency graphs.

Our comparison of the weighted and non-weighted union strategy and the optimization of union weights is a very first step towards the challenge of choosing an appropriate merging strategy. But as already mentioned in Section 4, there might exist much better combination strategies that we have not explored yet. The presented review on strategies others used in their studies at least provides a framework to systematically search for better solutions. Nonetheless, a comparative evaluation of combination strategies for code dependencies is still lacking.

6 Conclusion

The study shows a positive impact of evolutionary data on software clustering. A clustering exclusively based on evolutionary dependencies, however, is only successful if substantial evolutionary data is available. Evolutionary dependencies often do not cover the set of classes sufficiently. This seems to be the main reason for the better performance of the aggregated structural dependencies.

Thus, when clustering a system by only taking evolutionary dependencies into account, it is most important to rely on extensive historical data that covers the essential parts of the system. The main advantages over a purely structural based clustering are that also non-source files can be considered and that the approach works independent of the programming language (in our study a light-weight parser was just used to identify classes).

An integration of the two data sources unites the advantages of the approaches at the cost of a more complex data acquisition: In addition to a

parsed system core, the clustering approach is still able to handle non-source files and non-parsed source files. The clustering quality increases in our experiments, especially when strengthening duplicate dependencies. This confirms the assumption by Andritsos and Tzerpos (2005) that integrating evolutionary dependencies may have a positive impact on the clustering result. Related studies show similar improvements in other software engineering applications.

We filtered evolutionary dependencies successfully by confidence and support to increase the reliability of the dependencies. Thereby, filtering by confidence works better than filtering by support. A slight filtering turns out to be the best strategy when the clustering relies on evolutionary dependencies exclusively. A stronger filtering provides better results if evolutionary data is only an addition to structural data. Integrating unfiltered evolutionary dependencies bears the risk of decreasing the clustering quality.

These data-centered experiments demonstrate how important the choice and preprocessing of data sources in the domain of software clustering is. Also supported by the results from related studies, we believe that studying more data sources (e.g., dynamic dependencies, documentation, bug reports, software metrics), other preprocessing techniques, as well as different data combination strategies is essential for software clustering. Finally, the gained insights will help to tailor and customize software clustering techniques for particular applications like program comprehension, software (re)modularization, or software reuse.

References

Adams B, Jiang ZM, Hassan AE (2010) Identifying crosscutting concerns using historical code changes. In: ICSE '10: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ACM, New York, NY, USA, pp 305–314

Andritsos P, Tzerpos V (2005) Information-theoretic software clustering. IEEE Transactions on Software Engineering 31(2):150–165

Anquetil N, Fourrier C, Lethbridge TC (1999) Experiments with clustering as a software remodularization method. In: WCRE '99: Proceedings of the 6th Working Conference on Reverse Engineering, IEEE Computer Society, Washington, DC, USA, pp 235–255

Arafat O, Riehle D (2009) The comment density of open source software code. In: ICSE 09: 31st International Conference on Software Engineering - Companion Volume, IEEE, pp 195–198

Ball T, Kim JM, Porter AA, Siy HP (1997) If your version control system could talk ... In: ICSE '97 Workshop on Process Modeling and Empirical Studies of Software Engineering, ACM Press

Bavota G, De Lucia A, Marcus A, Oliveto R (2010) Software Re-Modularization based on structural and semantic metrics. In: WCRE '10: Proceedings of the 17th Working Conference on Reverse Engineering, IEEE Computer Society, pp 195–204

Beck F, Diehl S (2010a) Evaluating the impact of software evolution on software clustering. In: WCRE '10: Proceedings of the 17th Working Conference on Reverse Engineering, IEEE Computer Society, pp 99–108

Beck F, Diehl S (2010b) Visual comparison of software architectures. In: SoftVis '10: Proceedings of the ACM 2010 Symposium on Software Visualization, Salt Lake City, Utah, USA, pp 183–192

Beck F, Diehl S (2011) On the congruence of modularity and code coupling. In: ESEC/FSE '11: Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering, ACM, New York, NY, USA, pp 354–364

Beyer D, Noack A (2005) Clustering software artifacts based on frequent common changes. In: IWPC '05: Proceedings of the 13th International Workshop on Program Comprehension, IEEE Computer Society, pp 259–268

Bittencourt RA, Santos GJS, Guerrero DDS, Murphy GC (2010) Improving automated mapping in reflexion models using information retrieval techniques. In: WCRE '10: Proceedings of the 17th Working Conference on Reverse Engineering, IEEE Computer Society, pp 163–172

Breu S, Zimmermann T (2006) Mining aspects from version history. In: ASE '06: Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering, IEEE Computer Society, pp 221–230

Canfora G, Cerulo L, Di Penta M (2006) On the use of line co-change for identifying crosscutting concern code. In: ICSM '06: Proceedings of the 22nd IEEE International Conference on Software Maintenance, IEEE Computer Society, Washington, DC, USA, pp 213–222

Cataldo M, Mockus A, Roberts JA, Herbsleb JD (2009) Software dependencies, work dependencies, and their impact on failures. IEEE Transactions on Software Engineering 35(6):864–878

Cook TD, Campbell DT (1979) Quasi-Experimentation: Design & Analysis Issues for Field Settings. Houghton Mifflin

Fluri B, Gall HC, Pinzger M (2005) Fine-Grained analysis of change couplings. In: SCAM '05: Proceedings of the Fifth IEEE International Workshop on Source Code Analysis and Manipulation, IEEE Computer Society, Washington, DC, USA, pp 66–74

Gall H, Jazayeri M, Krajewski J (2003) CVS release history data for detecting logical couplings. In: IWPSE '03: Proceedings of the 6th International Workshop on Principles of Software Evolution, IEEE Computer Society, Washington, DC, USA

Gargiulo J, Mancoridis S (2001) Gadget: A tool for extracting the dynamic structure of Java programs. In: SEKE '01: Proceedings of the 13th International Conference on Software Engineering and Knowledge Engineering, pp 244–251

German DM, Hassan AE, Robles G (2009) Change impact graphs: Determining the impact of prior code changes. Information and Software Technology 51:1394–1408

Hansen KM, Jonasson K, Neukirchen H (2011) An empirical study of software architectures' effect on product quality. Journal of Systems and Software

84(7):1233–1243
Hattori L, Santos GD, Cardoso F, Sampaio M (2008) Mining software repositories for software change impact analysis: a case study. In: SBBD '08: Proceedings of the 23rd Brazilian symposium on Databases, Sociedade Brasileira de Computacao, Porto Alegre, Brazil, Brazil, pp 210–223
Kagdi H, Maletic JI (2007) Combining Single-Version and evolutionary dependencies for Software-Change prediction. In: MSR 07': Proceedings of the Fourth International Workshop on Mining Software Repositories, IEEE Computer Society, Washington, DC, USA
Kagdi HH, Gethers M, Poshyvanyk D, Collard ML (2010) Blending conceptual and evolutionary couplings to support change impact analysis in source code. In: WCRE '10: Proceedings of the 17th Working Conference on Reverse Engineering, IEEE Computer Society, pp 119–128
Kim S, Zimmermann T, Whitehead JE, Zeller A (2007) Predicting faults from cached history. In: ICSE '07: Proceedings of the 29th International Conference on Software Engineering, IEEE Computer Society, Washington, DC, USA, pp 489–498
Koschke R, Eisenbarth T (2000) A framework for experimental evaluation of clustering techniques. In: IWPC '00: Proceedings of the 8th International Workshop on Program Comprehension, IEEE Computer Society, Washington, DC, USA, pp 201–210
Kuhn A, Ducasse S, Girba T (2005) Enriching reverse engineering with semantic clustering. In: WCRE '05: Proceedings of the 12th Working Conference on Reverse Engineering, IEEE Computer Society, Washington, DC, USA, pp 133–142
Ma KL (2008) Stargate: A unified, interactive visualization of software projects. In: PacificVis '08: Proceedings of the IEEE VGTC Pacific Visualization Symposium 2008, pp 191–198
Maarek YS, Berry DM, Kaiser GE (1991) An information retrieval approach for automatically constructing software libraries. IEEE Transactions on Software Engineering 17(8):800–813
Mancoridis S, Mitchell BS, Rorres C, Chen Y, Gansner ER (1998) Using automatic clustering to produce high-level system organizations of source code. In: IWPC '98: Proceedings of the 6th International Workshop on Program Comprehension, IEEE Computer Society, Washington, DC, USA, pp 45–52
Mancoridis S, Mitchell BS, Chen Y, Gansner ER (1999) Bunch: A clustering tool for the recovery and maintenance of software system structures. In: ICSM '99: Proceedings of the IEEE International Conference on Software Maintenance, IEEE Computer Society, Washington, DC, USA, pp 50–59
Maqbool O, Babri HA (2007) Hierarchical clustering for software architecture recovery. IEEE Transactions on Software Engineering 33(11):759–780
Melton H, Tempero E (2007) The CRSS metric for package design quality. In: Proceedings of the thirtieth Australasian conference on Computer science, Australian Computer Society, Inc., Darlinghurst, Australia, ACSC '07, pp 201–210

Mitchell BS (2002) A heuristic approach to solving the software clustering problem. PhD thesis, Drexel University

Mitchell BS, Mancoridis S (2001) Comparing the decompositions produced by software clustering algorithms using similarity measurements. In: ICSM '01: Proceedings of the 17th IEEE International Conference on Software Maintenance, pp 744–753

Mitchell BS, Mancoridis S (2007) On the evaluation of the Bunch search-based software modularization algorithm. Soft Computing 12(1):77–93

Parnas DL (1972) On the criteria to be used in decomposing systems into modules. Communications of the ACM 15(12):1053–1058

Patel C, Lhadj AH, Rilling J (2009) Software clustering using dynamic analysis and static dependencies. In: CSMR 09: Proceedings of the 2009 European Conference on Software Maintenance and Reengineering, IEEE Computer Society, Washington, DC, USA, pp 27–36

Shtern M, Tzerpos V (2004) A framework for the comparison of nested software decompositions. In: WCRE '04: Proceedings of the 11th Working Conference on Reverse Engineering, IEEE Computer Society, Washington, DC, USA, pp 284–292

Shtern M, Tzerpos V (2007) Lossless comparison of nested software decompositions. In: WCRE '07: Proceedings of the 14th Working Conference on Reverse Engineering, IEEE Computer Society, Washington, DC, USA, pp 249–258

Sindhgatta R, Pooloth K (2007) Identifying software decompositions by applying transaction clustering on source code. In: COMPSAC '07: 31st Annual International Computer Software and Applications Conference - Vol. 1, pp 317–326

Stevens WP, Myers GJ, Constantine LL (1974) Structured design. IBM Systems Journal 13(2):115–139

Tzerpos V, Holt RC (1999) MoJo: A distance metric for software clusterings. In: WCRE '99: Proceedings of the 6th Working Conference on Reverse Engineering, IEEE Computer Society, Washington, DC, USA, pp 187–193

Vanya A, Hofland L, Klusener S, van de Laar P, van Vliet H (2008) Assessing software archives with evolutionary clusters. In: ICPC '08: Proceedings of the 16th IEEE International Conference on Program Comprehension, IEEE Computer Society, Los Alamitos, CA, USA, pp 192–201

Voinea L, Telea A (2006) CVSgrab: Mining the history of large software projects. In: EuroVis '06: Joint Eurographics - IEEE VGTC Symposium on Visualization, Eurographics Association, pp 187–194

Wen Z, Tzerpos V (2004) An effectiveness measure for software clustering algorithms. In: IWPC '04: Proceedings of the 12th International Workshop on Program Comprehension, pp 194–203

Wen Z, Tzerpos V (2005) Software Clustering based on Omnipresent Object Detection. In: IWPC '05: Proceedings of the 13th International Workshop on Program Comprehension, IEEE Computer Society, Washington, DC, USA, pp 269–278

Wierda A, Dortmans E, Somers LL (2006) Using version information in architectural clustering - a case study. In: CSMR '06: Proceedings of the Conference on Software Maintenance and Reengineering, IEEE Computer Society, Washington, DC, USA, pp 214–228

Wong S, Cai Y (2009) Predicting change impact from logical models. In: ICSM '09: IEEE International Conference on Software Maintenance, IEEE Computer Society, pp 467–470

Wu J, Hassan AE, Holt RC (2005) Comparison of clustering algorithms in the context of software evolution. In: ICSM '05: Proceedings of the 21st IEEE International Conference on Software Maintenance, IEEE Computer Society, Washington, DC, USA, pp 525–535

Xiao C, Tzerpos V (2005) Software clustering based on dynamic dependencies. In: CSMR '05: Proceedings of the 9th European Conference on Software Maintenance and Reengineering, IEEE Computer Society, Washington, DC, USA, pp 124–133

Zhou Y, Würsch M, Giger E, Gall HC, Lü J (2008) A bayesian network based approach for change coupling prediction. In: WCRE '08: Proceedings of the 15th Working Conference on Reverse Engineering, IEEE Computer Society, Washington, DC, USA, pp 27–36

Zimmermann T, Weißgerber P (2004) Preprocessing CVS data for fine-grained analysis. In: MSR '04: Proceedings of the 1st International Workshop on Mining Software Repositories, IEEE Computer Society, pp 2–6

Zimmermann T, Diehl S, Zeller A (2003) How history justifies system architecture (or not). In: IWPSE '03: Proceedings of the 6th International Workshop on Principles of Software Evolution, IEEE Computer Society, Washington, DC, USA

Zimmermann T, Weißgerber P, Diehl S, Zeller A (2004) Mining version histories to guide software changes. In: ICSE '04: Proceedings of the 26th International Conference on Software Engineering, IEEE Computer Society, pp 563–572