UNIVERSITÄT TRIER

Improving Software Clustering with Evolutionary Data

by

Fabian Beck

Diplomarbeit

Fachbereich IV
Informatik

February 2, 2009

Erklärung zur Diplomarbeit

Hiermit erkläre ich, dass ich die Diplomarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt und die aus fremden Quellen direkt oder indirekt übernommenen Gedanken als solche kenntlich gemacht habe. Die Diplomarbeit habe ich bisher keinem anderen Prüfungsamt in gleicher oder vergleichbarer Form vorgelegt. Sie wurde bisher auch nicht veröffentlicht.

Trier, den ______________

____

UNIVERSITÄT TRIER

Fachbereich IV
Informatik

Diplomarbeit

Improving Software Clustering with Evolutionary Data

by Fabian Beck

Abstract

The evolution of a software project is a rich data source for analyzing and improving the software development process. But does the information about how developers change the source code of a software systems also support to meaningfully group the elements of the software system?

Recently, some researchers have incorporated different kinds of evolutionary information into software clustering. Their results are promising but are not sufficient to finally assess the quality of evolution based software clustering because a direct comparison to common clustering approaches based on structural data is still missing. This thesis conducts several clustering experiments with an approved clustering algorithm on six different software projects comparing and combining evolutionary and structural data. These experiments show that evolutionary data produces competitive clustering results in many cases. Furthermore, evolutionary data is able to improve the quality of structural based clustering approaches. Returning to the above question, this work attests software evolution to be a valuable data source for software clustering.

Contents

List of Tables

Chapter 1

Introduction

The evolution of a software project is a rich data source. To analyze how developers change the source code of a software systems helps project leaders to control the development progress, helps software architects to detect design flaws, helps developers to find related files or hidden dependencies, and helps quality controllers to identify bugs. But does the evolution of a software project also support to meaningfully group the elements of the software system?

The current introduction gives an overview of related areas of research, namely software clustering and software evolution, and finally formulates the objectives and hypotheses of this thesis, which address the above question.

1.1 Software Clustering

The field of research that deals with the automatically grouping of software artifacts is called *software clustering*. Different software artifacts might serve as subjects: functions might be clustered to modules and classes [1, 55, 58, 63], or files to subsystems [2, 6], or classes to packages and components [10, 25, 40, 72]. Software clustering is an important discipline in software engineering in general, but especially in reverse engineering. It is used, for instance, to understand complex software systems [42, 43, 62], to (re-)modularize whole software architectures [5, 73], to identify reusable components [41], or to detect misplaced software artifacts [64].

Software clustering algorithms need information about the software artifacts to yield a reasonable clustering. Common clustering approaches usually retrieve this information directly from the static source code of the artifacts. Such structural data are, for example, method invocations and variable references among methods [4, 43, 46], or inheritance, aggregation, and method invocations among classes [10, 50, 72]. Some approaches try to improve the clustering by taking dynamic code dependencies collected during the program execution into consideration [30, 75]. Other approaches use the source code only indirectly by analyzing file names [6], variable names and comments [39, 42], or file ownerships [14, 4]. Recently, some researches already performed first clustering experiments based on the evolution of a software system—the according studies will be exhaustively introduced in Section 1.3. Although there exists such a rich variety of data sources, only few approaches integrate several of them into their clustering technique [4, 72].

Based on such relationship information among software artifacts, a clustering algorithm transforms the set of software artifacts into a decomposition of the software system. To this end, general data clustering algorithms can be applied: A common approach is to describe the entities to be clustered by a set of features. In the case of software clustering, for example, such a feature may be *extends class A*—the features are often binary attributes. The whole description of an entity is called *feature vector* and usually consists of a large number of dimensions. There exist diverse metrics to compute the similarity of two entities by comparing their feature vectors; Maqbool and Babri [46] introduce and compare several of them with respect to software clustering. A general clustering algorithm is the *agglomerative clustering algorithm*. It works bottom-up and thus starts with all entities in different clusters (singletons). In every iteration, the algorithm unites the two most similar clusters until only one cluster exists. The result is a hierarchy on the entities that can be transformed into a decomposition of the software system by cutting it at a certain level. In the first step of the algorithm, the similarities of the clusters are obviously equivalent to the similarities of the entities. In the further steps, however, a similarity measure for arbitrary clusters (with more than one element) is needed. Common approaches are, for example, to take only the two most similar elements of the two clusters into account (*single linkage*), to consider the two most dissimilar (*complete linkage*), or to compute an average similarity over all pairs of elements (*average linkage*).

Nevertheless, algorithms exist that are specialized for the task of clustering software artifacts. Andritsos and Tzerpos [4] propose such a specialized software clustering approach, called *LIMBO*, that aims to minimize the loss of information (in terms of the feature vector) in every step of the clustering. It uses a similarity metric derived from information theory and a modified version of the previously described general clustering algorithm. Other works employing such feature data models can be found in [5, 39, 66]. Moreover, approaches exist that are based on totally different data models. For instance, the clustering tool *Bunch* [24, 44] works on a graph structure instead of a feature based data model: the software artifacts are represented as nodes that are directly connected by dependencies (edges). The clustering algorithm of *Bunch* rests upon the principle that the coupling between software artifacts from different clusters should be low while the cohesion among software artifacts from the same cluster should be high—a well-established concept of a good code architecture, already introduced in 1974 [59]. *Bunch* optimizes the software decomposition to meet this requirement. Tzerpos and Holt [62] introduce another graph based approach. Their clustering algorithm, called *ACDC*, is searching for typical subsystem patterns in the dependency graph. They argue that the resulting clusters will be more comprehensible for users. Other clustering techniques are also based on graph data models [11, 20, 42, 52, 53]. Which clustering technique is particularly going to be used in the current work will be discussed in Section 2.4.

1.2 Software Evolution

Software evolution, i.e., studying the development process of software projects, is a growing research field in software engineering. Diehl [23] distinguishes two major areas within this field of research: First, *Design for Change* focuses on researching development patterns and software designs that assist future changes. Second, *Analysis of Software Histories* deals with detecting interesting facts in the change history of a software project. Clustering the artifacts of a software system based on its history certainly belongs to the latter category.

The evolution of a software project is documented, among other things, by its release history or, more fine-grained, by the changes applied to its source file during the development. In modern software engineering these changes are stored in a revision control system such as the popular systems *CVS* and *SVN*, here simply referred to as a *version archive*. Since it is the developer's decision when to commit his changes, the system is not able to track every atomic change, it just stores the committed versions. Jointly committed changes by the same developer form a transaction. There exist many tools that prepare and visualize this evolutionary revision data, for instance, in an aggregated overview [28, 19], in an animation [21], or on a time axis [17, 18, 29, 65]. Such visualizations are used to explore and to understand the software system [17, 18, 21, 65], or to detect architectural weaknesses [28]. Beyond visualizing the evolutionary data, change transactions from version archives are used, e.g., for change recommendation [79], error prediction [37], or refactoring detection [68].

When two software artifacts have often been changed together during the development, one can assume that they are somehow related. This relation is called *evolutionary dependency* (or *evolutionary/co-change coupling*). Several similar exact definitions of this concept exist [9, 28, 77]. Evolutionary dependencies might be useful to cluster software artifacts because frequent common changes indicate strong implicit dependencies (which might not be detectable in the source code). Visualizing these evolutionary dependencies as a graph structure already enables to detect clusters and outliers manually: node-link [28, 19] as well as matrix [19] representations can be employed for this task.

1.3 Integrating Software Clustering and Software Evolution

Recently, researchers started to link ideas from both fields of research, software clustering and software evolution. Depending on their research background, their approaches either focus on enriching a software clustering algorithm with basic evolutionary information or applying a software clustering algorithm on exclusively evolutionary data.

1.3.1 Clustering-Centered Approaches

Andritsos and Tzerpos [3, 4] enrich the structural feature vectors (derived from source code) of their already introduced clustering technique *LIMBO*: They evaluate how additional non-structural information like the file ownership, the directory path, the number of lines of code, and the timestamp of the last file change influences the clustering result. The results of their study identify the ownership and directory information to be valuable to improve the clustering. Although the quality of the clustering stays nearly uninfluenced by the timestamp information, the authors guess that a more elaborate change information (i.e., a better evolutionary data source) promises additional improvements. They also introduce and evaluate different weighting schemes on feature vectors. Their experiments, however only performed on structural data, show that some weighting schemes are able to improve the clustering quality.

Another idea is that the intended architecture of a software system might be represented in a purer form in the initial version than in a later one. Wierda et al. [72] tried to use this effect for the recovery of the system architecture by combining different versions. The structural dependencies of one version of the system are represented as a dependency graph: two dependency graphs are merged by an intersection or union operation. Finally, the authors use the *Bunch* clustering algorithm. Their evaluation on two example systems shows that combining the current version with the first version by intersection improves the decomposition significantly, whereas combining it with the previous version or using the union operation does not provide better results.

1.3.2 Evolution-Centered Approaches

Other researchers are focusing on the evolutionary aspect: In one of the first works on software evolution, Ball et al. [9] already used a specialized graph layout algorithm on an evolutionary dependency graph. Clusters emerge as visual groups in the graph visualization. Beyer and Noack [11, 12] refine this approach: Instead of a dependency graph they employ a graph that relates transactions with their participating files. The graph is visualized using a specialized energy-based graph layout that intends to reveal clusters. To improve readability, only file nodes are displayed, transaction nodes and edges are hidden. These clustering approaches based on graph visualization, however, need a user to finally mark the clusters in the visualization and are suffering from occlusion and non-planarity problems.

Vanya et al. [64] identify clusters in a software system with a more traditional clustering approach. They model each transaction as a feature, compute file similarities using these features, and finally perform an average linkage clustering algorithm on the similarity matrix. By comparing the resulting evolutionary decomposition to the current architecture of the software, they are able to identify design flaws in the software architecture. In a case study, experts for the studied software project rated most of the information about detected design flaws as valuable.

Voinea and Telea [65, 66] integrate a clustering algorithm based on software evolution into their tool *CVSgrab*, which visualizes the evolution of a software system. Every file of the system is represented as vertical bar, and the evolutionary clustering is used to group similar files. In contrast to the previously described approach by Vanya et al., the agglomerative clustering algorithm of *CVSGrab* does not work on the transaction data of the version archive. Instead it works on pure commit moments: The similarity of two files is defined by the neighborhood in time of the commits. The authors argue that this approach—unlike a transaction based approach—is able to correlate files that have different owners.

Finally, Bowman and Holt [14] investigate the relations between subsystem dependencies of a software system in different views. They compare dependencies in the conceptual architecture to dependencies in the concrete architecture and to dependencies in the ownership architecture (it considers files that are developed by the same person as dependent). Although no clustering algorithm is applied, the evaluation results show that this form of evolutionary information—the ownership dependencies—can be used as a predictor for structural dependencies.

1.4 Objectives and Hypotheses

The previously introduced methods that integrate software clustering and software evolution are very promising. They show

- that software clustering based on structural data can be improved by integrating evolutionary data sources like file ownership or by intersecting two versions of the software and
- that it is possible to cluster software only by using evolutionary data with graph layout algorithms or traditional clustering algorithms.

But despite these positive results, there are some aspects that are not covered sufficiently yet:

- The clustering-centred approaches only employ very basic evolutionary data: file ownership or the first and latest program version. There exist, however, no studies that estimate the value of integrating the rich data source of transaction based evolutionary dependencies.
- In contrast, the evolution-centered approaches use this data source. But although they show that their clustering techniques are working to some degree, they do not allow to assess the value of the clustering results with respect to structural data sources.

The goal of the present thesis is to overcome these shortcomings by comparing structural and evolutionary data sources directly to each other and by integrating both to improve the overall clustering result. The positive results of the described recent studies suggest the following very general hypothesis.

Hypothesis 1. *Software evolution provides valuable information for software clustering.*

This first hypothesis can be confirmed in different ways: The evolutionary data might provide competitive clustering results compared to the structural data, or the evolutionary data might be able to improve a structural clustering result. Since the integration of basic evolutionary data already resulted in positive effects on the clustering results, the following second hypothesis is allowed to refine the general statement of Hypothesis 1.

Hypothesis 2. *It is possible to improve current software clustering algorithms that use structural information by integrating evolutionary data.*

Moreover, the evidence from the previously conducted studies is not enough to predict that software clustering exclusively based on evolutionary data can be successful in general. Nevertheless, it will be interesting to observe the qualities of such clustering results.

This work conducts an extensive study to test the hypotheses. Chapter 2 introduces an appropriate experimental design that is able to cluster software systems—based on structural data, evolutionary data, as well as combined data—and to assess the results. The study is focusing on two major applications of software clustering: architecture recovery and architecture improvement. Based on these applications Chapter 3 and Chapter 4 present the actual experiments and analyze their results. Finally, Chapter 5 summarizes and concludes the findings.

Chapter 2

Experimental Design

After introducing the sample projects used in the evaluation, this chapter describes the general experimental design step by step: The required dependency information has to be extracted from different data sources and transformed into a comparable form. Then, a clustering algorithm can be applied on this data. The quality of the resulting clustering decompositions is used to compare the two data sources. An integrated software environment implements all these steps.

2.1 Terminology

Object-oriented software systems consist of classes and interfaces. While classes and interfaces have to be handled differently on programming level, they are very similar from the more abstract perspective of architecture: Inheritance is realizable with classes as well as with interfaces. Furthermore, both can use another class or interface in their definition. The only difference on this level of abstraction is that classes can aggregate other classes and interfaces as attributes while interfaces can neither aggregate classes nor other interfaces. Summing up, classes and interfaces use comparable concepts on the architecture level, and thus, the term *class* represents a class or an interface in the present thesis if not explicitly indicated otherwise.

2.2 Sample Software Projects

Every study on software clustering needs to select at least one sample software project that serves as a subject in the experiments. The current work employs the following set of open source *Java* projects:

- *Apache Tomcat* [7], a *Java Servelet* implementation,
- *Azureus* [8] (now called *Vuze*), a *BitTorrent* file sharing client,
- *JEdit* [32], a text editor,
- *JFreeChart* [33], a *Java Swing* chart library,
- *JFtp* [34], an *FTP* client, and
- *JUnit* [36], a regression testing framework.

Weißgerber already used these projects in his refactoring detection evaluation [67]. The present author chose them for practical reasons—because their data has been available to the author in a preprocessed form—and for theoretical reasons—because the selection includes a broad spectrum of software types: The spectrum ranges from user clients and libraries to server applications. Although this set of projects cannot be considered statistically representative for the whole population of software projects, it is representative to a certain extent as it covers a wide range of project types.

Table 2.1 shows detailed information about the selected programs. Their version archives provide the necessary evolutionary data for the present study. As version archive systems, both popular systems, *CVS* and *SVN*, are supported. The program versions were not chosen intentionally but were determined by the checkout date in spring 2007—an exception forms the *JFtp* project. The numbers of classes (based on the latest version) give an idea of the project sizes: *JFtp* is the smallest examined project with only 78 classes while *JEdit* is the largest one with 840 classes. Note that the *Azureus* and *Tomcat* project are each restricted to one of their main packages because they were originally larger than 1000 classes—more than the experimental environment that will be presented in the following is able to handle efficiently. Finally, the number of transactions and developers (obtained from usernames) provides some further information of the software development process: For example, although the *JFtp* project is much smaller than the *JFreeChart* project (in number of classes as well as transactions), both projects have the same number of developers (five).

2.3 Data Sources

The present work does not focus on evaluating the software clustering algorithm itself but on evaluating the information the clustering algorithm is working with. Thus, the input data of the algorithm is the independent variable in the experiments. The necessary

Table 2.1: Characteristic data of the sample software projects and their repositories.

Project	Archive	Time frame	# Classes	# Transactions	# Developers
Azureus	CVS	2003/07/10 – 2007/02/14	477[a]	10665	27
JEdit	SVN	2001/09/02 – 2007/02/12	840	2190	20
JFreeChart	CVS	2001/10/18 – 2007/02/14	794	2413	5
JFtp	CVS	2002/01/25 – 2003/03/23	78	210	5
JUnit	CVS	2002/12/12 – 2007/02/08	317	673	7
Tomcat	SVN	2006/03/27 – 2007/03/10	561[b]	661	13

[a]restricted to `org.gudy.azureus2.core3`
[b]restricted to `org.apache.catalina`

data is provided by structural information from the source code and by evolutionary information from the version archive.

As discussed in Section 1.1, there exist different approaches in the domain of software clustering to implement such information in a data structure. Here, a graph based approach, where the nodes represent the classes and the edges represent the dependencies between classes, is favored over a feature based approach. It is chosen because this data structure implements class dependencies directly and fits the clustering algorithm that is preferred for the present experiments as it will be argued in Section 2.4. Anquetil and Lethbridge [5] compare these two approaches and also highlight the *appealing simplicity* of the graph data model. But they concurrently argue that the feature data model has a higher flexibility(which is, however, not necessary in this work) and tends slightly to produce better clustering results (but that largely depends on the employed clustering algorithm).

2.3.1 Structural Dependencies

Structural dependencies are dependencies that are derived by exclusively analyzing the source code of a software project. As classes of an object-oriented system are the entities to be clustered, this work discerns the following three types of structural class dependencies:

Inheritance The concept of inheritance is one of the main concepts to relate two classes in object-oriented programming. Class A depends on class B through inheritance if class A extends class B.

Aggregation An alternative to incorporate the functionality of another class is to define it as an attribute (a variable on class level). To keep it simple, modifiers like `static`, `final`, or `private` are ignored as well as the multiplicity of the aggregation. Thus, class A depends on class B through aggregation if class B is needed to define any attribute of class A.

Usage The most common form of inter-class dependencies, however, is the simple usage of another class in a method or constructor (e.g., as a local variable, as a method parameter, or by a method invocation). Class A depends on class B through usage if class A uses class B in a method or constructor.

The source code of the sample software systems has to be analyzed to extract all necessary information about these concepts. Since *Java* is based on polymorphism, the real, dynamic dependencies can only be detected during the program execution. For example, class A uses class B as a method parameter, and class B1 extends class B. It may be possible that class A receives and uses class B1 through a method call at runtime although no direct dependency between class A and class B1 can be detected in the source code. Nevertheless, in such cases the dependency exists indirectly in the static source code through a combination of dependencies (in the example through usage and inheritance). As a detection of the dynamic dependencies would be very complex, the present work has to focus on static structural dependencies.

In *Java*, classes are usually defined in a source code file that only contains that single class. But classes can also be defined in a context of another class as member classes (named classes defined on class level), local classes (named classes defined in a method), or anonymous classes (unnamed classes defined in a method) (e.g., [26]). As member and local classes can be identified by their class name and can be extracted from their containing class easily, they are handled like other classes. In contrast, anonymous classes will be ignored because they can be neither identified nor extracted easily. Moreover, interfaces are treated like classes as already discussed in Section 2.1.

The following formalism introduces the foundations to define a formal data model for structural dependencies; it adopts the style of the definitions used in the *Unified Framework for Coupling Measurement* by Briand et al. [15]. Speaking of a particular software system like in the following definition, the present work always refers to the latest checked-out version of the system as listed in Table 2.1.

Definition 2.1 (Children, Attributes, Usage)**.** *Consider an object-oriented software system S at a certain version. Let $C(S)$ be the set of all classes of S (according to the preliminary remarks).*

*(a) The **children** of a class $c \in C(S)$ are the set* $\mathrm{Children}(c) \subset C(S)$ *of classes that directly extends c.*

(b) *The **attributes** of a class $c \in C(S)$ are the set $A(c) \subset C(S)$ of classes that are used to define the directly implemented attributes in class c.*

(c) *The **usage** of classes is described by the set $\text{Uses}(c) \subset C(S)$ of classes that are directly used in a method or constructor of c.*

Based on these definitions, a graph data structure can be formulated for each of the structural dependency kinds. Since none of the structural relations is symmetric, the appropriate graph types are directed graphs.

Definition 2.2 (Class Inheritance Graph)**.** *The directed graph*

$$\begin{aligned} G_{\text{CIG}} &:= (V_{\text{CIG}}, E_{\text{CIG}}) \\ V_{\text{CIG}} &:= C(S) \\ E_{\text{CIG}} &:= \{(c_1, c_2) : c_1 \in \text{Children}(c_2)\} \end{aligned}$$

*is called **Class Inheritance Graph** (*CIG*).*

Definition 2.3 (Class Aggregation Graph)**.** *The directed graph*

$$\begin{aligned} G_{\text{CAG}} &:= (V_{\text{CAG}}, E_{\text{CAG}}) \\ V_{\text{CAG}} &:= C(S) \\ E_{\text{CAG}} &:= \{(c_1, c_2) : c_2 \in A(c_1)\} \end{aligned}$$

*is called **Class Aggregation Graph** (*CAG*).*

Definition 2.4 (Class Usage Graph)**.** *The directed graph*

$$\begin{aligned} G_{\text{CUG}} &:= (V_{\text{CUG}}, E_{\text{CUG}}) \\ V_{\text{CUG}} &:= C(S) \\ E_{\text{CUG}} &:= \{(c_1, c_2) : c_2 \in \text{Uses}(c_1)\} \end{aligned}$$

*is called **Class Usage Graph** (*CUG*).*

To integrate the three concepts of structural coupling, a combined graph can be defined as follows:

Definition 2.5 (Structural Class Dependency Graph)**.** *The directed graph*

$$\begin{aligned} G_{\text{SCDG}} &:= (V_{\text{SCDG}}, E_{\text{SCDG}}) \\ V_{\text{SCDG}} &:= C(S) \\ E_{\text{SCDG}} &:= E_{\text{CIG}} \cup E_{\text{CAG}} \cup E_{\text{CUG}} \end{aligned}$$

*is called **Structural Class Dependency Graph** (*SCDG*).*

To construct these graphs finally from source code, the tool *DependencyFinder* [22] is employed. *DependencyFinder* is a code analysis suite that works on compiled *Java* bytecode and, among other things, is able to extract all relevant dependencies into an *XML* file. As the compiled bytecode is usually not included in the version archive, the experimenter has to compile the source code of the latest considered sample project version manually.

The *GraphML* format [31] is used to store the graphs persistently because it is both flexible and *XML*-based. An *XSLT* script, developed by the author, transforms the dependencies that are contained in the *DependencyFinder XML* output file into the *GraphML* format according to the definitions given above for the CIG, CAG, and CUG. The SCDG is generated later on.

2.3.2 Evolutionary Dependencies

For evolutionary analyses of software projects not only the current version of the project but also previous versions must be available. Thus, version archives provide a perfect data source for such analyses. In this thesis, evolutionary dependencies between classes are determined as follows: Class A depends on class B through evolution if class A is often changed *together* with class B.

As two classes cannot really be edited concurrently by the same developer, it is unclear what the term *together* exactly means in this context. Version archives do not document every change of a file, only sets of changes made by a developer and finally committed concurrently to the archive. These sets are called transactions.

Definition 2.6 (Transaction)**.** *For a software system S in a certain version, let $T_i \subset C(S)$ be the i^{th} set of changed classes that are concurrently submitted to the version archive (by a single author and with the same log message). T_i is called the i^{th}* ***transaction*** *of the version archive.*

Thus, class A and class B are changed *together* if they are members of the same transaction. Note that only classes that are also contained in the latest version are considered ($T_i \subset C(S)$) because dependencies among classes that no longer exist are not interesting for clustering. Since classes are identified by their name and package, renamed or moved classes cannot be tracked. All evolutionary information about the respective classes before the rename or move refactoring is lost.

Based on the definition of transactions, it may be possible to define a pair of classes as evolutionary dependent if they are at least once member of the same transaction.

But some transactions might relate files randomly, for example, if a developer fixed two totally unrelated bugs in two different files. Hence, a mechanism that allows to filter out such noise and considers only strong dependencies might improve the reliability and thus the quality of the evolutionary data. Zimmermann et al. [77] introduce the concept of support and confidence to measure the strength of evolutionary dependencies. The support value of a dependency counts how often the linked software artifacts were changed together, i.e., were part of the same transaction. Additionally, the confidence value of a dependency relates the support to the total number of changes applied to one of the artifacts. According to the definition of Zimmermann et al. [77], this concept is formalized as follows.

Definition 2.7 (Evolutionary Support and Confidence)**.** *Let $c_1, c_2 \in C(S)$ be two classes and $\{T_i\}_{i=1}^{l}$ a sequence of transactions.*

(a)

$$\text{Support}(c_1, c_2) := |\{T_i : c_1 \in T_i, c_2 \in T_i\}| \in \mathbb{N}$$

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

(b) If c_1 is element of at least one transaction T_i,

$$\text{Confidence}(c_1, c_2) := \frac{\text{Support}(c_1, c_2)}{\text{Support}(c_1, c_1)} \in [0, 1]$$

*is called **confidence** of the evolutionary dependency of class c_1 to class c_2. Otherwise,* $\text{Confidence}(c_1, c_2) := 0$.

In the definition of Confidence the value of $\text{Support}(c_1, c_1)$ represents the total number of transactions in which c_1 is changed. Thus, the value of Confidence is 1 (the maximum confidence) if c_2 is always changed when c_1 is changed. Note that Support is a symmetric function whereas Confidence is not a symmetric function.

Support and Confidence yield a pair values for two classes, c_1 and c_2. It is difficult to handle such a tuple as a measure for inter-class dependencies. The following definition introduces a simplified concept that allows to classify all pairs of classes into a set of dependent and independent pairs of classes.

Definition 2.8 (Evolutionary Dependency)**.** *In an object-oriented system S, the set*

$$\text{EvDependent}_{\alpha}^{k}(c) := \{c' \in C(S) : \text{Support}(c, c') > k \wedge \text{Confidence}(c, c') > \alpha\}$$

*of a class $c \in C(S)$ with parameters $\alpha \in [0, 1)$ (minimum confidence) and $k \in \mathbb{N}$ (minimum support) is called the **set of evolutionary dependent classes** for class c. And a tuple of classes (c_1, c_2) is called **evolutionary dependency** if $c_2 \in \text{EvDependent}_{\alpha}^{k}(c_1)$.*

Since Confidence is not symmetric, EvDependent is also not symmetric in terms of $c_2 \in \text{EvDependent}^k_\alpha(c_1) \not\Rightarrow c_1 \in \text{EvDependent}^k_\alpha(c_2)$.

Analogously to the structural dependencies, an evolutionary dependency graph can be defined with the help of the EvDependent sets. But in contrast to the previous graphs, the evolutionary graph depends on parameters because EvDependent does: the minimal confidence α and the minimal support k. Thus, a reasonable parameter setting will have to be found in the experiments.

Definition 2.9 (Evolutionary Class Dependency Graph)**.** *The directed graph*

$$\begin{aligned}
G_{\text{ECDG}^k_\alpha} &:= (V_{\text{ECDG}^k_\alpha}, E_{\text{ECDG}^k_\alpha}) \\
V_{\text{ECDG}^k_\alpha} &:= C(S) \\
E_{\text{ECDG}^k_\alpha} &:= \{(c_1, c_2) : c_2 \in \text{EvDependent}^k_\alpha(c_1)\}
\end{aligned}$$

is called ***Evolutionary Class Dependency Graph*** *(*ECDG*) with parameters* $\alpha \in [0, 1)$ *(minimum confidence) and* $k \in \mathbb{N}$ *(minimum support)—short:* ECDG^k_α.

Finally, those evolutionary class dependency graphs are to be extracted from the version archives. As the archives are operating on file level and a *Java* class is not equivalent to a file in general, a more fine-grained analysis approach is necessary that is able to detect changed *Java* classes. Weißgerber and Zimmermann [67, 78] describe such an analysis for *CVS* archives of *Java* projects: It first parses general information from the *CVS log*, then restores the single transactions, and finally maps the changes to fine-grained artifacts like classes and methods with the help of a light-weight *Java* parser. All this information is stored in an *SQL* database for fast access. The second step, the restoration of transactions, is necessary because the concurrent check-in of several files is not registered as one transaction by the *CVS* system. A heuristic has to be used that groups all nearly concurrent changes (based on a sliding time window approach) with the same log message that have been committed by the same developer as one transaction. In [67], Weißgerber also introduces an equivalent approach for *SVN* archives, which needs not to use a transaction restoration because *SVN* systems preserve the transaction information. He kindly provided me the current version of his *Java* library that implements these techniques for *CVS* as well as *SVN* archives.

Based on this library, I developed a converter that reads the transaction data of the sample software projects and exports a class dependency graph with support and confidence values as the edge weights. It became evident in practice not to store the ECDG in different parameter settings directly, but instead, to store the raw data (the support and confidence values) and apply the filtering later on. The transformation process omits large transactions (here, transaction with more than 50 participating classes) to reduce noise in the evolutionary dependency data. The goal of this technique is to filter out transactions that produce random dependencies (e.g., the initial check-in transaction or transactions with global refactorings) and is widely used in the domain of software evolution (e.g., [76, 78]). To avoid conflicts because of several copies of the same class, the converter also ignores classes from branched versions, i.e., it only works on the trunk of the archive.

Although it is possible to relate non-source files with the concept of evolutionary dependency, the present work is restricted to source code files to guarantee the comparability to structural dependencies. But in real world applications the possibility to cluster non-source files might be a crucial advantage of evolutionary dependencies over structural dependencies.

2.3.3 Graph Operations

The definitions of the structural and evolutionary graphs provide a comparable data structure to evaluate the performance of structural versus evolutionary data sources for software clustering. Nevertheless, a mechanism to integrate both data sources is still missing.

All introduced graphs are based on the same set of nodes (the classes of the software system) while the dependencies are different. But the dependencies are also represented in a comparable form, as edges. Including all dependencies in a single graph would provide the desired integration. Thus, the integration can be formulated as a simple union operation on graphs.

Definition 2.10 (Graph Union Operation)**.** *Given two unweighted directed graphs, $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, the **graph union operation** $\cup$ applied to G_1 and G_2 creates an unweighted directed graph*

$$G_1 \cup G_2 := G_3 = (V_3, E_3)$$

with $V_3 := V_1 \cup V_2$ and $E_3 := E_1 \cup E_2$ (here, $\cup$ identifies the normal set union operation).

As the present work only compares and combines graphs that belong to the same software system S, the equation $V_1 = V_2 = V_3 = C(S)$ will hold true in every application of the union operation. Note that the SCDG is equivalent to CIG $\cup$ CAG $\cup$ CUG.

After the union operation is applied, the relevance of all dependencies is identical—it is not evident to which graph a dependency originally belonged. As one data source might be more reliable than another, an important information is lost. To preserve this information, three groups of dependencies have to be distinguished:

(a) dependencies that belong to the first original graph but not to the second,

(b) dependencies that belong to both graphs, and

(c) dependencies that belong to the second original graph but not to the first.

These three groups of dependencies can be represented in a single graph by assigning an importance value to each group. The importance value is implemented as an edge weight for all members of the group. This technique can be expressed as a weighted union operation on two unweighted graphs that results in a weighted graph.

Definition 2.11 (Weighted Graph Union Operation)**.** *Given two unweighted directed graphs, $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, the* ***weighted graph union operation*** $\cup_{[\omega_a,\omega_b,\omega_c]}$ *with $\omega_a, \omega_b, \omega_c \in \mathbb{R}$ applied to G_1 and G_2 creates a weighted directed graph*

$$G_1 \cup_{[\omega_a,\omega_b,\omega_c]} G_2 := G_3 = (V_3, E_3, \mu)$$

with $V_3 := V_1 \cup V_2$, $E_3 := E_1 \cup E_2$, and a weight function $\mu : E_3 \to \mathbb{R}$ defined as

$$\mu(e \in E_3) := \begin{cases} \omega_a & \textit{if } e \in E_1 \wedge e \notin E_2 \\ \omega_b & \textit{if } e \in E_1 \wedge e \in E_2 \\ \omega_c & \textit{if } e \notin E_1 \wedge e \in E_2 \end{cases}$$

Note that, while the simple union operation is symmetric, the weighted union operation is not symmetric, except if $\omega_a = \omega_c$. Moreover, the definition allows weights of 0, which will result in ignoring the according dependencies in the clustering.

2.4 Clustering Algorithm

The clustering algorithm is the core of the present experiments: If the clustering algorithm does not work properly, the clustering results cannot show meaningful differences between structural and evolutionary data. Thus, it is important to choose a good clustering algorithm.

As discussed in Section 1.1, various software clustering approaches have been proposed and studied. Since this work does not aim at improving a particular clustering algorithm directly but at assessing the quality of different data sources, any good clustering algorithms would serve the purpose. It is neither necessary nor helpful to develop yet another clustering algorithm.

To improve the generalizability, it would be desirable to use different clustering algorithms. Nevertheless, the present studies focus on one clustering algorithm and leave a study with other clustering algorithms to future work because several clustering algorithm would multiply the effort of creating an appropriate experimental environment.

2.4.1 Clustering Decomposition

In general, a clustering algorithm divides a set of entities (here, classes) into several groups (the clusters); the result of this division is called clustering decomposition. Depending on the algorithm, the decomposition is either flat or hierarchical: A flat decomposition is a simple partition of the entity set. A hierarchical decomposition is a sequence of partitions starting with a simple partition, which is summarized step by step until all entities are included in one set. Formally, a decomposition can be defined as follows.

Definition 2.12 (Decomposition)**.** *Let $C(S)$ be the set of classes of a software system S.*

(a) A partition $\mathcal{P}$ of $C(S)$, i.e.,

$$\mathcal{P} = \{P_1, \dots, P_k\} \subset 2^{C(S)}, \quad \bigcup_{i=1}^{k} P_i = C(S), \quad P_i \cap P_j = \emptyset \ \forall i \neq j,$$

is called ***flat decomposition*** *of the software system S. $\mathfrak{P}_{C(S)}$ is the set of all flat decompositions of the software system S.*

(b) A family of partitions $\overline{\mathcal{P}} = \{\mathcal{P}^1, \dots, \mathcal{P}^l\}$ where for every level m $(1 < m \leq l)$ each set $P_i^m \in \mathcal{P}_m$ can be represented as $P_i^m = \bigcup_{j \in J} P_j^{m-1} \neq \emptyset$ for an index set J and $\mathcal{P}_l = \{C(S)\}$ is called ***hierarchical decomposition*** *of the software system S. $\overline{\mathfrak{P}}_{C(S)}$ is the set of all hierarchical decompositions of the software system S.*

2.4.2 Choosing a Clustering Algorithm

To justify the choice of the clustering algorithm that will be used in the present work, the following paragraphs refer to related evaluations on software clustering. These evaluations compare different software clustering algorithms, mainly the ones already introduced in Section 1.1, in diverse experimental setups.

The authors of *LIMBO*, Andritsos and Tzerpos, present a comparative study on software clustering with two sample software projects [3]. The study includes the *LIMBO*, *ACDC*, and *Bunch* algorithm as well as complete, single, and average linkage approaches. In [70] and [4] the study is repeated with more evaluation metrics and an additional sample projects. Summarizing the very similar results of these experiments, the *LIMBO* algorithms yields better results than the other algorithms (which cannot be ordered further because of close results).

Wu et al. [74] compare the clustering results of six clustering algorithms (complete and single linkage, each in two setups; *ACDC*; *Bunch*) applied to five sample software projects. They measure the clustering quality on three dimensions over long sequences of monthly checked-out versions:

Stability If the clustering results that were produced after small changes are similar to the ones produced before, the algorithm is stable. The single linkage algorithms and *ACDC* perform best while *Bunch* produces the most unstable results.

Authoritativeness If the clustering is similar to a good reference decomposition, the authoritativeness is high. Here, the complete linkage approach and *Bunch* produce the best results.

Extremity of Clustering Distribution If the clustering algorithm avoids to produce extreme clusters, like a single huge cluster or many tiny clusters, the extremity is low (i.e., good). While the single linkage algorithms, *ACDC*, and one of the complete linkage algorithms produce extreme decompositions, the results of *Bunch* are far better in terms of more evenly distributed clustering sizes.

In summary, *Bunch* performs best in this study for it has good results on the dimensions of authoritativeness and extremity; the low stability is caused by the random component of *Bunch* and can be eased by repetitive clustering runs.

Recently, Maqbool and Babri conducted a study where they compared hierarchical clustering algorithms on four sample software systems [46]. The studied algorithms are namely complete linkage, single linkage, average linkage, *LIMBO*, and the so-called *Combined Algorithm* [54] and *Weighted Combined Algorithm* [45]. Since they proof that a variant of the *Weighted Combined Algorithm* is equivalent to *LIMBO*, the two algorithms are not evaluated separately. The study includes several assessment dimensions: Although *LIMBO* (respectively the *Weighted Combined Algorithm*) is less stable than the other algorithms, it produces better decompositions compared to a set of decompositions proposed by experts.

These evaluations indicate that, first of all, *LIMBO* but also *Bunch* produce good clustering results. *LIMBO* is a feature based clustering approach that focuses on the minimization of information loss while *Bunch* is graph based and follows the concept of low coupling and high cohesion. The present work will employ *Bunch* because its graph based approach realizes the inter-class dependencies more directly: A dependency is represented as an edge in the graph whereas in a feature based approach entities would be clustered because of similar features (directly dependent classes need not have similar features). Nevertheless, *LIMBO* could be chosen as well—it would be very interesting to repeat the present study with this feature based algorithm.

2.4.3 *Bunch*

This section presents the selected clustering algorithm *Bunch* in detail. *Bunch* is a clustering approach that optimizes a clustering quality metric with a heuristic search technique and produces hierarchical clustering decompositions. It is implemented in the *Java* programming language and can be used either as a stand-alone application or as a library. Besides the papers [24, 44, 43] by Mancoridis, Mitchell et al., the PhD thesis of Mitchell [47] provides an exhaustive description of the *Bunch* clustering tool and its *Java* API. Moreover, *Bunch* is freely available [16]. The following brief introduction of *Bunch* summarizes the works of Mancoridis, Mitchell et al. and describes all important parameters of *Bunch*. Since it is not possible to vary these parameters systematically in every part of the study, a reasonable default setup has to be chosen (it is summarized in Table 2.2).

Bunch works on a graph structure that is called *Module Dependency Graph*. The graph represents modules as nodes and module dependencies as directed edges. As the terms *module* and *dependency* are not bound to a strict definition, the dependency graphs defined in Section 2.3 can be considered as a *Bunch Module Dependency Graph*: classes are equivalent to modules, and structural or evolutionary dependencies are equivalent to module dependencies. In the default setting of *Bunch* the edges are not weighted, but it is also possible to use weighted edges as required for the graphs produced by the weighted union operation.

To optimize a decomposition, *Bunch* needs a metric to estimate the quality of the current decomposition. The underlying quality criterion is the concept of low coupling and high cohesion between modules/classes in a software system. The edges of the dependency graph are partitioned into inter-edges (edges with source and target in the same cluster) and intra-edges (edges with source and target in different clusters). *Bunch* provides a metric using this classification of edges in three implementations: *BasicMQ* is a straight implementation of the principle of low coupling and high cohesion but has a high computational complexity. *TurboMQ* is an efficient heuristic of *BasicMQ* but can be further accelerated through an incremental computation, referred to as *ITurboMQ*. The only reasonable choice is to use *ITurboMQ*, the fastest metric.

Using weighted graphs, I had problems with the class `bunch.TurboMQIncrW` of the *Bunch* system, which implements the *ITurboMQ* metric: the metric evaluation showed no differences for non-weighted and weighted graphs. Thus, I re-implemented this class according to the documented specification in [47] (Chapter 3.4.3) and validated my implementation using a small test data set by comparing the results to manually evaluated metric values.

Bunch implements three clustering methods, which differ in the kind of optimization strategy they use to find good clustering decompositions in terms of high quality metric values.

Exhaustive Search Algorithm This algorithm performs an exhaustive search by systematically checking all possible solutions.

Hill Climbing Algorithm Starting with a random decomposition the algorithm searches for a better solution in the neighborhood. The neighborhood consists of all decompositions that can be derived by one elementary transformation. This is repeated for the improved solution until a local maximum is reached.

Genetic Algorithm A set of random decompositions forms an initial population. A genetic algorithm using selection, reproduction, crossover, and mutation operations aims to improve the decomposition quality generation by generation.

The exhaustive search algorithm is only applicable to systems with a very small number of modules because of its computational complexity and thus cannot be used in the study. Furthermore, the genetic algorithm tends to yield results of unstable quality in varying runtime. The hill climbing algorithm, however, produces stable high quality results with predictable runtime. Thus, the present work prefers this algorithm.

Using the hill climbing algorithm, further parameters need to be set. I performed some performance tests for the *JFtp* project to find a good setup.

Population Size The quality of the local maximum reached by the algorithm might be bad compared to the global maximum. To increase the probability to find a high quality clustering, the algorithm can be applied to a set of random initial decompositions (the population) instead of just starting with one initial decomposition. Although it sounds reasonable to choose a population larger than 1, the tests showed that this would just increase the runtime without any significant effect on the clustering quality.

Search Space This parameter defines how many percent of the neighboring decompositions should be at least evaluated searching for a better solution. The test results do not significantly change for varying search space parameter settings. Hence, the parameter is set to its default value of 0, i.e., the first decomposition that is better than the current one is chosen (*nearest ascent hill climbing*).

Extensions Advanced extensions of the hill climbing algorithm, like *building blocks* or *simulated annealing*, are omitted because they also showed no eminent clustering quality improvements.

A study on the performance of *Bunch* by Mitchell and Mancoridis [50] confirms that neither larger population sizes nor simulated annealing increases the clustering quality. The other parameters are not tested in their study.

Table 2.2: Employed setup of the clustering tool *Bunch*.

Parameter	Value
Modularization quality metric	*ITurboMQ*
Clustering algorithm	Hill climbing
Population size	1
Search space	0
Algorithm extensions	off

Bunch produces hierarchical decompositions of a software system although the search heuristic and the quality metric work on flat decompositions only: The clustering algorithm is recursively applied on an aggregated *Module Dependency Graph* that treats each detected cluster as a single node and unites the edges accordingly. The recursion stops when all nodes are included in one cluster. The decompositions produced by the steps of this algorithm form a hierarchy with the final single cluster as the root element.

Since *Bunch* uses a heuristic search approach that has a random element, the whole clustering method can be considered as a random experiment. Mitchell and Mancoridis showed in their study about *Bunch* [50] that the resulting decompositions mostly differ only slightly. Nevertheless, several repetitions of the experiment will be performed to increase the stability of the results.

2.5 Evaluation Method

To determine the quality of a software clustering technique, the quality of the resulting decompositions must be assessed. A first approach is to review the decomposition of one or more software systems manually to get insights in the pros and cons of the examined technique. This approach is often used in literature (e.g., [12, 39, 43]) and is easy to realize, but the evidence is limited due to its high subjectiveness and a weak scalability. A more elaborate and more objective approach is to estimate the quality of a software decomposition with the help of some standardized metric.

The software decomposition can be evaluated on the one hand by considering some internal quality criteria, on the other hand by comparing it to a reference decomposition that is known as being a high quality decomposition. In the present work, according to Maqbool and Babri [46], these approaches are referred to as internal and external assessment.

For internal assessment, different metrics that map a decomposition to a quality value can be employed: For example,

- the size and number of the clusters can be taken into account [5, 46],
- a high stability of the clustering result features a good clustering algorithm [46, 60], or
- low coupling and high cohesion indicates the clustering quality [5, 50] (like employed in *Bunch*).

For external assessment, the first aspect is the availability of a reference decomposition, which is often manually created by an expert. Secondly, the decompositions resulting from the automatic clustering can be compared to this reference decomposition. Hence, a metric is used to measure the similarity or distance between the two decompositions.

I decided to use an external instead of an internal assessment approach in my experiments for several reasons: Metrics like cluster size, number, and stability just give a hint at the clustering quality but do not independently measure the quality. For instance, a clustering algorithm that always includes all classes in a single cluster achieves a perfect result in terms of stability but obviously does not produce reasonable decompositions. Nevertheless, a metric based on cohesion and coupling, like the already available metric computed by *Bunch*, can be employed as an independent quality measure. But as the metric value largely depends on the input graph, the results are not comparable for different graphs, that is, the clustering results for structural and evolutionary dependency graphs cannot be compared directly. In contrast, an external assessment can be employed as an independent quality measure that does not use the dependency graph. There are, however, some other difficulties, which will be discussed in the following sections.

2.5.1 Reference Decompositions

In general, the external assessment refers to an assessment method that uses some kind of consensus as a benchmark for the clustering result. This benchmark is referred to as an *reference decomposition* of the software system (e.g., [49]), other terms are *expert decomposition* [5, 46], *authoritative decomposition* [3], or *reference corpus* [38].

The approach of creating a reference decomposition as a benchmark assumes that a perfect clustering exists, which is closely approximated by the reference decomposition. As the quality of a decomposition depends on the developer's opinion, there are no objective and reliable standards to determine the perfect clustering. Thus, this approach can only be considered a heuristic to estimate the quality of a software clustering method. The process of generating a reference decomposition can be implemented in different ways. These approaches are presented in the following and are summarized in Table 2.3.

The simplest method is to use the factual architecture of the system, for example, the package structure of an object-oriented system. This decomposition, which is called *factual decomposition*, uses the grown structure of the project generated by many experts—the software architects and developers—during the development process. Of course, a software architecture can degenerate over time, but one can assume that in a successful software project—like the studied programs in the present work—this loss of architectural quality is limited to a certain extend and that the architecture can still be considered a very good decomposition.

Possibly, a better quality can be achieved by employing one ore more independent experts to manually create the decomposition because they are not trapped in the historic structures of the project. But this does not guarantee to obtain an improved reference decomposition because their expertise might not be sufficient. If several experts are attending, the result of their work can be a set of decompositions or just one decomposition as a consensus. In comparison to the factual decomposition, however, these approaches result in highly increased effort for generating the decompositions.

A totally different approach is the automatic generation of a reference decomposition as described by Mitchell and Mancoridis [49]. They propose a framework called *CRAFT* (*Clustering Results Analysis and Tools*), which is able to detect common patterns in decompositions produced by different clustering algorithms. Thus, several algorithms instead of several experts create a consensus decomposition. While the quality of the expert decomposition is limited by the expertise of the experts, it is now limited by the quality of the clustering methods. This might be the main drawback of the *CRAFT* framework because one cannot really improve a state-of-the-art automatic clustering technique by employing another just as well automatic clustering technique as a reference.

Table 2.3: Summarized comparison of different methods to generate reference decompositions.

	(1) Factual decomposition	(2) Expert decomposition	(3) Set of expert decompositions	(4) Clustered decomposition
Description	Actual architecture of the project	One decomposition created by one or more experts	Set of decompositions created by several experts	Consensus of several clustering algorithms
Used in	[5, 43, 74]	[3, 38]	[46]	[49]
Quality	Medium	Medium, up to good	Medium, up to very good	Low
Effort	Very low	Medium, up to high	High, up to very high	Low (if clustering tools are available)

In the present thesis the quality of clustering decompositions will be evaluated for several software projects, which means that a reference decomposition must be generated for every project. The effort is a crucial factor, and thus, constructing a set of expert decompositions has to be excluded. Furthermore, the automatic generation of the decomposition is not suitable because of the questionable clustering quality of other clustering techniques. Finally, comparing the first and second approach, the expected, slightly improved quality of the second one does not justify time and effort for manually creating a new, high quality decomposition of every considered system. Hence, the factual decomposition is chosen as the most suitable method. Even the drawback of the possibly low quality of a single reference decomposition is relativized by the fact that several systems will be analyzed. The risk of a poor decomposition is distributed among all projects.

2.5.2 Similarity Measures for Decompositions

The decompositions created by the clustering algorithm have to be compared to the reference decomposition. The most similar decomposition is considered the best result. But this comparison is a non-trivial task because there is no natural metric that measures the similarity between two decompositions.

As Wen and Tzerpos pointed out in their study on such similarity measures [71], the measures can be grouped by their underlying data model: Two decompositions can be compared based on

- only the nodes of the dependency graph,
- only the edges of the dependency graph, or
- both the nodes and the edges of the dependency graph.

Considering the dependencies might be a good choice for evaluating different clustering algorithms on the same input data. But when comparing different input data, the quality metric cannot rely on the dependencies that are changing with the input data. Thus, only metrics based on nodes can be employed in the present work.

Another aspect of a clustering similarity metric is whether it works with flat or hierarchical decompositions. As described in the following, there exist many approaches comparing flat decompositions whereas metrics directly considering hierarchical decompositions are rare.

Anquetil and Lethbridge [5] propose to measure the precision and recall of intra pairs, i.e., pairs of entities that are in the same cluster: The precision value is the percentage of intra pairs in the clustering decomposition that are also intra pairs in the reference decomposition. And vice versa, the recall value is the percentage of intra pairs in the reference decomposition that are also intra pairs in the clustering decomposition. Although this method provides detailed information about the difference of both decomposition, it is difficult to use it as the single quality criterion because precision and recall must be reasonably related and combined to a single value.

A similarity measure based on cluster overlapping between clustering decomposition and reference decomposition is introduced by Koschke and Eisenbarth [38]: They classify overlapping clusters into two categories: the category *GOOD* includes pairs of clusters that are very similar, whereas the category *OK* includes pairs of clusters where one component is (at least nearly) a subset of the other component but not concurrently vice versa. Together with the set of components in the reference decomposition that are neither fully nor partly matched, called *true negatives*, they are able to define a recall value that estimates to which extent the components of the reference decomposition are retrieved by the clustering result. This approach, however, ignores *false positives*, that is, components of the clustering result that have no equivalent in the reference decomposition. Other drawbacks of the approach are its behavior in extreme cases and its disregard of necessary merge operations [71].

Tzerpos and Holt [61] developed a metric, called *MoJo*, that is estimating the distance between two decompositions with the minimal number of *Move* and *Join* operations needed to transform one decomposition into the other. Furthermore, Wen and Tzerpos introduced an optimal algorithm for the computation of *MoJo* [69] and an improved metric normalization [70]. The new metric, which is called *MoJoFM*, ranges from 0, representing a clustering that is farthest away from the reference, to 100, representing a clustering that is completely identical to the reference. The number of *Move* and *Join* operations is a natural measure to some extent because they are similar to operations a user would perform to transform one decomposition into the other. Moreover, the metric is clear and simple to understand and interpret.

While all these approaches require flat decompositions, the only yet published methods comparing hierarchical decompositions are the *END* framework [56] and the *UpMoJo* metric [57], both by Shtern and Tzerpos. *END* is a generic framework that reuses an arbitrary similarity metric for flat decomposition by comparing the decompositions level by level with this similarity metric and summing up the weighted results. It stays unclear, however, which underlying similarity metric and which weighting function should be employed. In contrast, *UpMoJo* is an independent similarity measure, not a generic framework. It extends the *MoJo* measure by a third operation, the *Up* operation, which allows to move up single entities or whole subsystems in the hierarchy by one level. The distance between two hierarchical decompositions is again represented by the number of operations to transform one decomposition into the other.

EdgeSim and *MeCl* by Mitchell and Mancoridis [48], and *EdgeMoJo* by Wen and Tzerpos [71] are further similarity metrics. But they cannot be applied in this study because they are based on the dependency relations.

At last, the *Precision/Recall* metric, the *Koschke-Eisenbarth* metric, the *MoJoFM* metric, the *END* framework, and the *UpMoJo* metric are possible candidates for an appropriate metric in the current use case. Since both hierarchical approaches, *END* and *UpMoJo*, are not evaluated in an extensive study yet and would add more complexity to the results of the studies, the present thesis focuses on the metrics for flat decompositions. The question how to transform a hierarchical decomposition into a flat one is discussed later. Since the interpretation of the *Precision/Recall* metric is unclear and the *Koschke-Eisenbarth* metric is not so comprehensible and ignores false positives, *MoJoFM*—a clear and simple metric—seems to be a good choice.

2.5.3 Method Realization

The effort to retrieve the factual architecture of a sample software system as the reference decomposition is low: The necessary information about the package structure is also included in the dependency *XML* file, which is generated by the *DependencyFinder* tool and is used to generate the structural dependency graphs (Section 2.3.1). A simple *XQuery* script extracts the package structure as the factual decomposition.

As already mentioned, *MoJoFM* depends on the minimum number of *Move* operations (a class is moved from one cluster to another) and *Join* operations (two clusters are joined to one cluster) which can be formalized as follows.

Definition 2.13 (Minimum Number of Move and Join Operations)**.** *For a software system S* ***the minimum number of Move and Join operations*** *that is needed to transform a flat decomposition $A \in \mathfrak{P}_{C(S)}$ into a flat decomposition $B \in \mathfrak{P}_{C(S)}$ is defined by a function*

$$\begin{aligned} \text{mno} : \mathfrak{P}_{C(S)} \times \mathfrak{P}_{C(S)} &\rightarrow \mathbb{N}_0^+ \\ (A, B) &\mapsto \text{mno}(A, B) \end{aligned}$$

Based on that definition the *MoJoFM* metric is introduced as a normalized asymmetric similarity function of decomposition A and decomposition B.

Definition 2.14 (MoJoFM)**.** *For two flat decomposition $A, B \in \mathfrak{P}_{C(S)}$*

$$\begin{aligned} \text{MoJoFM} : \mathfrak{P}_{C(S)} \times \mathfrak{P}_{C(S)} &\rightarrow [0, 100] \\ (A, B) &\mapsto 100 \cdot \left(1 - \frac{\text{mno}(A, B)}{\max_x(\text{mno}(x, B))}\right) \end{aligned}$$

is the ***MoJoFM*** *distance from A to B.*

As there is no inverse operation of the join operation, the function mno is not symmetric, that is, $\text{mno}(A, B) = \text{mno}(B, A)$ is not valid in general. Thus, *MoJoFM* is also not symmetric, contrary to the intuitive understanding of the term *similarity metric*.

The *MoJoFM* function is normalized to a range of $[0, 100]$ because $\text{mno}(A, B)$ is divided by $\max_x(\text{mno}(x, B))$, the maximum of operations to transform any decomposition x into partition B. 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. Thus, a low *MoJoFM* value close to 0 for A to B means that many operations are needed to transform A into B—the similarity is low—whereas a high value close to 100 means that only few operations are needed to transform A into B—the similarity is high.

But despite the normalization of *MoJoFM*, one cannot directly compare the *MoJoFM* values of different sample projects because the normalization is not an overall standard. But as $\max_x(\text{mno}(x, B))$ depends on the structure of B, the normalization is restricted to a certain reference decomposition. Thus, *MoJoFM* values can only be compared for the same reference decomposition B. If the reference decompositions are different, the information is, however, not completely worthless: It still gives a hint at the overall similarity to the reference decompositions.

As a flat decomposition is needed for the evaluation of the clustering with *MoJoFM*, a reasonable method for transforming the hierarchical decomposition retrieved from the package structure as well as from *Bunch* into a flat decomposition must be specified. Since a comparison to a very simple decomposition would be meaningless, it is desirable for the reference decomposition to keep as much information as possible. Thus, the most detailed level of the hierarchical decomposition is chosen. For the clustering result, however, one cannot choose a certain hierarchy level because this might favor one of the data sources. The simplest way to avoid this problem is just to compare the resulting hierarchy to the reference hierarchy for each level and to select the best *MoJoFM* value. Though, this best fit approach is not usable in real world applications because there, a level must be chosen without knowing the clustering quality. With the help of the experiments, however, a generally good level could be found.

2.6 Software Environment

Up to now, we have discussed the different parts of the experimental design including the sample projects, the data sources, the clustering algorithms, and the evaluation method. All together, these parts form the foundations of the present study on software clustering. Nevertheless, a meta application that is integrating the fragments into a working experimental environment is still missing. I developed a *Java* program called *Software Clustering Analysis Suite* (*SCAS*) that implements such an integration and, together with some additional tools and converters, forms the required experimental environment.

Briefly summarized, this environment should be able to import class dependency data from different data sources, transform the data into dependency graphs, cluster the classes based on the dependency information, and finally analyze the quality of the clustering results. Figure 2.1 shows this process and the underlying software architecture as a simplified diagram. External tools and data sources, which were not self developed, are colored gray.

2.6.1 Preprocessing

To obtain the dependency graphs, the dependency information must be extracted from the data sources. The preprocessing is different for structural and evolutionary data. It is pictured in the upper half of Figure 2.1, split into two columns, one for each of the data sources.

The source code of the latest checked-out version of the sample project is the base of the structural dependency graph. As discussed in Section 2.3.1, the external tool *DependencyFinder* (Version 1.2.0, [22]) is used to obtain the relevant inheritance, aggregation and usage information from the compiled source code. It is executed over its command line interface with its default settings. The resulting *XML* file contains much more information than the necessary dependencies. I developed an *XSLT* template to filter and transform the relevant dependency information to the *Class Inheritance Graph* (CIG), the *Class Aggregation Graph* (*GAG*), and the *Class Usage Graph* (CUG), all stored in the *GraphML* format.

Additionally, an *XQuery* script creates a reference decomposition from the *DependencyFinder XML* file: To this end, the script iterates over all classes and assigns each class to a cluster identified by its aggregating package. This decomposition of the sample system is output into a text file to be later read for the *MoJoFM* computation. Thus, the decomposition forms the required flat reference decomposition that matches the current system architecture.

For the evolutionary dependencies, my export tool uses the *Java* library provided by Peter Weißgerber [67] as described in Section 1.3.2: The export tool iterates over all transactions skipping large transactions and branches and collects all dependency information for the computation of the support and confidence values. The resulting evolutionary dependency graph is exported into the *GraphML* format, but not exactly according to the definition of the *Evolutionary Class Dependency Graph* (ECDG, Definition 2.9): Instead of applying several filters for support and confidence (resulting in separately stored dependency graphs), the raw data—support and confidence values—are handled as edge weights. The necessary filtering has to be performed by *SCAS* later on while importing the evolutionary dependencies.

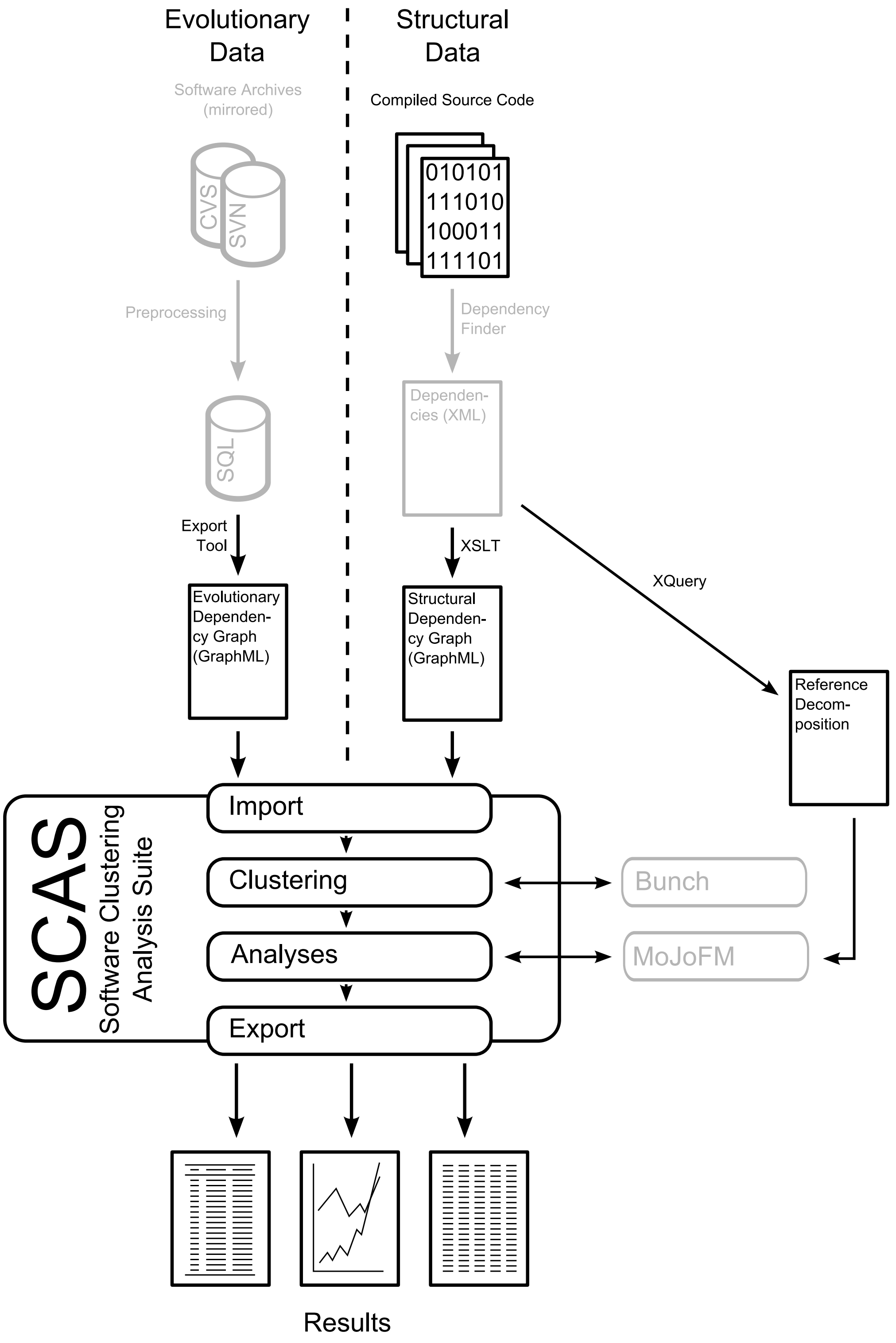

FIGURE 2.1: Simplified architecture of the experimental environment. External tools are colored gray.

2.6.2 Software Clustering Analysis Suite

The *Software Clustering Analysis Suite* (*SCAS*) is the main component of the software environment. Although *SCAS* might at first glance look like a set of converters, it actually addresses more tasks:

- It provides a dependency graph data structure that allows to perform various operations on the graphs,
- parallelizes the time consuming clustering runs,
- summarizes the clustering results in complex data tables,
- and finally manages the workflow of the clustering experiments in different setups.

SCAS provides an import module for the *GraphML* format to read the dependency graphs. It transforms the graphs to an internal graph representation that uses and extends the graph data structure provided by the *JGraphT* library (Version 0.7.3, [35]). For example, one of the extensions is the implementation of the union operations as defined in Section 2.3.3. Such a union on the simple structural graphs is performed to create the *Structural Class Dependency Graph* (SCDG). *SCAS* also applies the omitted filtering by support and confidence to import the *Evolutionary Class Dependency Graph* (ECDG) matching exactly its definition. Depending on the experiment setup, further transformations or operations like uniting graphs with the weighted union operation might be necessary. Finally, the internal graph structure includes all needed graphs for every sample software project.

It might be interesting to learn some facts about the dependency graphs. To this end, *SCAS* provides some analysis features like assessing basic characteristics (number of nodes and edges) or intra-package edges (dependencies that relate classes in the same package).

Now, the imported and transformed graphs can be clustered with the *Bunch* tool (Version 3.3.6, [16]): Although *Bunch* is available as a *Java* library, the graph data cannot be passed to *Bunch* as a parameter, but has to be available in a simple text based graph format. Thus, the graphs are exported again and clustered with the *Bunch* tool using the setup specified in Section 2.4.3. Depending on the graph size, a single clustering run with *Bunch* needs less than second up to a few minutes on an up-to-date PC. Due to the repetitive runs, however, this becomes the main performance bottleneck of the experiments. To use the full computation power of a multi-processor core system, the clustering task is split into different threads: every repetitive clustering run is outsourced to its own thread, i.e., there exist as many threads as clustering repetitions and each thread has exactly the same task. In some cases, the employed version of *Bunch* freezes for an unknown reason while performing the clustering. Then, *SCAS* has to cancel and restart the respective *Bunch* thread. Finally, the clustering results can be accessed over the *Bunch* API.

To measure the quality of the clustering decompositions, the *MoJoFM* metric of the *MoJo Java* library (Version 2.0, [51]) is employed. Similar to *Bunch*, the *MoJo* library is directly controlled by *SCAS*, but the raw data, here the clustering and reference decomposition, has to be passed per text file. The *Bunch* API provides the clustering decomposition while the previously generated reference decomposition is already available. *SCAS* summarizes the *MoJoFM* results in data tables that can be exported, for example, as LaTeX tables.

SCAS is implemented with a command line interface whereas the current configuration is stored in a properties file. Each properties file represents a certain clustering experiment. *SCAS* is started with such a properties file as a parameter: It reads the properties, runs the experiment according to the properties, and finally output the results on the command line. For example, properties are the structural graph to use, the filtering to be applied on evolutionary graphs, the union operations to be performed, the kind of clustering to be run, or the structure of the result table.

Chapter 3

Study on Architecture Recovery

One of the main use cases of software clustering is architecture recovery. When the architecture of a software system is totally undocumented or the documentation is just outdated, software clustering helps to retrieve the current architecture information. Furthermore, an automatically recovered architecture supports the developers to remodularize a badly structured project. For *Java* projects this applies, for example, if all classes are placed in the default package or if the package structure is not thoroughly maintained in a growing project. Thus, architecture recovery is an important application for software clustering.

The first study of the present work addresses the task of architecture recovery. It evaluates the clustering quality in terms of similarity to the factual architecture by using the factual decomposition as a reference for the *MoJoFM* similarity metric as described in Section 2.5. The study is split into several experiments that stepwise test the hypotheses formulated in Section 1.4. Thereby, the experiments aim to improve the clustering results.

3.1 Graph Characteristics

Before discussing the actual clustering experiments, Table 3.1 presents characteristics of the dependency graphs that help to understand the future clustering results. It portrays the graphs in terms of number of nodes and edges.

The table shows that the selected sample projects are significantly varying in size, measured by the number of nodes: the smallest project, *JFtp*, has just 78 nodes (i.e., classes), in contrast to the largest project, *JEdit*, which has 840 nodes. Please recall that *Azureus* and *Tomcat* are each restricted to one of their main packages because the experimental

Table 3.1: Dependency graph characteristics in number of nodes and edges.

		Azureus	*JEdit*	*JFreeChart*	*JFtp*	*JUnit*	*Tomcat*
# Nodes (classes)		477	840	794	78	317	561
# Edges (dependencies)	CIG	279	318	484	40	133	312
	CAG	434	691	223	66	80	455
	CUG	2269	4020	3714	38	851	2207
	SCDG	2362	4117	3937	127	864	2348
	$\mathrm{ECDG}^0_{0.0}$	6218	29372	13922	1184	2216	696
	$\mathrm{ECDG}^0_{0.2}$	1354	8438	9512	738	1814	502
	$\mathrm{ECDG}^0_{0.4}$	727	4011	5427	414	1318	293
	$\mathrm{ECDG}^0_{0.6}$	345	2117	2478	261	975	177
	$\mathrm{ECDG}^0_{0.8}$	277	1655	2063	166	920	169
	$\mathrm{ECDG}^0_{0.99}$	267	1627	2058	158	920	169
	$\mathrm{ECDG}^1_{0.0}$	2330	9430	2376	530	218	84
	$\mathrm{ECDG}^1_{0.2}$	650	3289	1991	365	170	63
	$\mathrm{ECDG}^1_{0.4}$	302	1542	1334	251	134	45
	$\mathrm{ECDG}^1_{0.6}$	172	888	930	171	91	25
	$\mathrm{ECDG}^1_{0.8}$	104	426	515	76	36	17
	$\mathrm{ECDG}^1_{0.99}$	94	398	510	68	36	17
	$\mathrm{ECDG}^2_{0.0}$	1258	4538	696	322	56	24
	$\mathrm{ECDG}^2_{0.2}$	419	1713	588	231	43	23
	$\mathrm{ECDG}^2_{0.4}$	193	814	338	164	22	18
	$\mathrm{ECDG}^2_{0.6}$	99	390	172	122	15	13
	$\mathrm{ECDG}^2_{0.8}$	51	247	56	50	5	11
	$\mathrm{ECDG}^2_{0.99}$	41	219	51	42	5	11
	$\mathrm{ECDG}^3_{0.0}$	786	2798	330	202	26	10
	$\mathrm{ECDG}^3_{0.2}$	288	1137	300	159	24	10
	$\mathrm{ECDG}^3_{0.4}$	129	504	178	111	6	6
	$\mathrm{ECDG}^3_{0.6}$	63	256	82	84	3	5
	$\mathrm{ECDG}^3_{0.8}$	35	155	31	37	0	3
	$\mathrm{ECDG}^3_{0.99}$	25	127	26	29	0	3

environment is not able to handle more than 1000 classes efficiently. Hence, the projects range from small to mid-size projects.

Comparing the simple structural dependency graphs with respect to their edge density, it is obvious that the *Class Usage Graph* (CUG) is far more dense than the *Class Inheritance Graph* (CIG) and the *Class Aggregation Graph* (CAG). The number of edges of the two latter graphs, CIG and CAG, does not significantly exceed the number of nodes in each project: The two graphs have an average edge degree of about 1 or lower. This is a first hint that the clustering might be hard only using inheritance or aggregation dependencies because an average edge degree of 1 or lower indicates that there must be many nodes without any dependency information—it is impossible to cluster these nodes. The CUG, however, is promising because it contains about three times more dependencies than nodes.

The difference in number of edges between the SCDG and the CUG is small. Thus, most of the dependencies in the SCDG are usage dependencies. Moreover, the difference is even significantly smaller than the number of edges of the CAG and the CIG. Thus, many of the inheritance and aggregation dependencies are already included in the usage graph. This is likely because inherited and aggregated classes will be probably used in the inheriting or aggregating class. Though, it will be interesting to observe in the following experiments whether the minor difference between the CUG and the SCDG has an impact on the cluster quality.

As you may recall from Section 2.3, the *Evolutionary Class Dependency Graph* (ECDG) depends on two threshold values, the minimum support and the minimum confidence. These two parameters can be considered as filters that are getting stronger (i.e., reducing the number of dependencies) with increasing values. Since it is hard to define reasonable threshold values in advance without knowing any clustering results, the first experiments will vary support and confidence systematically in a grid pattern: support threshold values are $(0, 1, 2, 3)$ and confidence threshold values are $(0.0, 0.2, 0.4, 0.6, 0.8, 0.99)$. The last confidence value of 0.99 is chosen instead of 1.0 because a confidence threshold of 1.0 would exclude every dependency as, according to Definition 2.8 and 2.9, the confidence values must be greater than and not equal to the threshold value. Furthermore, the $\mathrm{ECDG}^{0}_{0.0}$ represents the unfiltered graph, i.e., its edge set includes all pairs of classes that are changed together at least once. The systematical variation finally results in 24 ECDG graphs.

Table 3.1 lists the number of edges for these evolutionary graphs and confirms that the number is reduced block by block for increasing support values and line by line for increasing confidence values. One can assume that a good clustering result is only possible without or with just a moderate filtering because for stricter filters the number of edges drops to an edge degree near 1 or below (i.e., there are many nodes without any dependency information). Although these strictly filtered graphs probably will yield poor clustering results when they are exclusively used as the data source, they might be very helpful for enhancing structural data sources because the filtering is supposed to increase the reliability of the dependencies.

It is also interesting to compare the evolutionary dependencies by the sample projects: For example, examining the unfiltered graph, $\mathrm{ECDG}^{0}_{0.0}$, the number of dependencies for *JFtp* (1184) is greater than for *Tomcat* (696) although *JFtp* (78 classes) is much smaller than *Tomcat* (561 classes). In contrast, the *Azureus* project, which has about the same size as *Tomcat* in number of classes, has a much more dense $\mathrm{ECDG}^0_{0.0}$ (6218 dependencies). Hence, the evolutionary dependencies for *Tomcat* are very sparse and it will be hard to cluster *Tomcat* with these dependencies exclusively. Furthermore, the filtering affects the projects differently: Comparing the similarly sized *Azureus* and *JUnit* project while increasing the filter threshold, the number of dependencies drops much faster for a rising confidence threshold for *Azureus* than for *JUnit*. And for a rising support threshold it is the other way around. Thus, a compromise has to be found that yields good clustering results for at least most projects.

3.2 Clustering with Simple Data Sources

The first experiment addresses the question whether it is possible to get high quality clustering decompositions using only single 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. The clustering is performed as described in Chapter 2 by the *SCAS* tool using *Bunch* as the clustering algorithm and *MoJoFM* as the quality metric. Table 3.2 presents the results of the experiment as *MoJoFM* metric values for each clustering setup. As *Bunch* yields different results for repetitive runs on the same input data, the clustering runs have to be repeated to increase the precision of the quality information. The *MoJoFM* values in Table 3.2 are averaged over $n = 50$ runs, which is a compromise between precision and runtime.

A precision measure of a mean value $\overline{x}$ is the standard error (e.g., [13]). It is computed by distributing the variance of the measured variable (here, the *MoJoFM* values) to the n repetitions.

Definition 3.1 (Estimated Standard Error of the Mean)**.** *Let $\hat{\sigma}^2$ be the estimated population variance. Then, the* ***estimated standard error of the mean*** *is defined as*

$$\hat{\sigma}_{\overline{x}} = \sqrt{\frac{\hat{\sigma}^2}{n}}$$

As the notation already implies, the standard error can be interpreted as the standard deviation of the mean. Since the mean value of n identical distributed independent random variable is approximately a Gaussian variable (central limit theorem), the mean value is located in an interval $[\overline{x} - \hat{\sigma}_{\overline{x}}, \overline{x} + \hat{\sigma}_{\overline{x}}]$ with an estimated probability of 68% (e.g., [13]). Table 3.2 also shows the standard errors of the means (in the notation $\pm\hat{\sigma}_{\overline{x}}$).

Additional to the clustering results for the structural and evolutionary graphs, the table lists a default quality metric value in the last row: This value represents the quality of a virtual decomposition where each class is located in the same cluster, i.e., the decomposition consists of only one huge cluster. This decomposition, however, needs not be the worst clustering with respect to the required number of move and join operations. Thus, the *MoJoFM* quality of this default decomposition is normally greater than zero. The value is added to the table to provide a reference for the *MoJoFM* values of a project: Although *MoJoFM* is normalized, *MoJoFM* values of different projects are not directly comparable because the normalization largely depends on the structure of the reference decomposition (see Section 2.5.3). For Table 3.2 that means that *MoJoFM* values can only be compared vertically but not horizontally. With the default decomposition as a reference, however, one can roughly estimate the overall quality of a single decomposition.

3.2.1 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 the hypothesis that the CIG and CAG usually do not contain enough information for a competitive clustering result. There exist, however, some exceptions: For *JUnit* the quality of the CIG, CUG, and CAG results are nearly equal although the CUG is much denser than the CIG and CAG. And for *JFtp* the CIG (53.4) even outperforms the CUG (49.4) although the density of the CIG and CUG is nearly equal (Table 3.1). Thus, the comparably good clustering quality in both cases must be the result of a better dependency data quality of the CIG and CAG than of the CUG. This quality relation might even hold for the other projects but might be concealed by the disadvantage of the comparably sparse inheritance and aggregation information.

As an aggregation of the CIG, CAG, and CUG, the SCDG unites the information from the three simple structural graphs and hopefully increases or at least steadies the clustering quality of the simple data sources. The average clustering quality of the SCDG (54.3) shows this effect because it is clearly higher than the CIG and CAG average values (41.9 and 37.9) and at least slightly higher than the CUG value (53.2). Discerning by projects, the clustering quality of the CUG, however, is slightly higher in three of six cases. This is probably the outcome of the minimal difference of the CUG and SCDG. Nevertheless, for *JFreeChart* and *JFtp* the SCDG clearly outperforms the CUG. Thus, the SCDG can be considered a stabilized version of the CUG.

TABLE 3.2: *MoJoFM* clustering quality based on single structural and evolutionary dependency graphs. For structural as well as evolutionary graphs, the best overall quality values are highlighted in light gray and the best project specific ones in gray.

	Azureus	*JEdit*	*JFreeChart*	*JFtp*	*JUnit*	*Tomcat*	Average
CIG	30.2±0.2	42.0±0.5	35.4±0.3	53.4±0.4	58.0±0.1	32.8±0.1	41.9
CAG	28.6±0.3	47.1±0.7	14.3±0.0	45.7±0.3	55.9±0.0	35.8±0.0	37.9
CUG	52.4±0.4	63.5±1.7	40.6±0.6	49.4±0.0	58.8±0.2	54.5±0.5	53.2
SCDG	49.9±1.0	65.5±1.6	44.2±0.4	54.0±0.3	58.4±0.2	54.0±0.5	54.3
$\text{ECDG}^0_{0.0}$	31.7±0.1	40.4±0.1	40.7±0.2	41.5±0.2	60.5±0.1	11.1±0.0	37.7
$\text{ECDG}^0_{0.2}$	34.7±0.4	46.2±0.4	41.3±1.0	40.0±0.2	60.0±0.0	11.1±0.0	38.9
$\text{ECDG}^0_{0.4}$	35.6±0.1	47.0±0.2	43.2±0.1	42.3±0.4	61.9±0.1	11.3±0.0	40.2
$\text{ECDG}^0_{0.6}$	33.6±0.2	47.2±0.1	40.9±0.1	42.4±0.3	61.5±0.0	11.4±0.0	39.5
$\text{ECDG}^0_{0.8}$	32.6±0.3	45.9±0.1	39.8±0.1	42.7±0.3	60.0±0.0	11.4±0.0	38.7
$\text{ECDG}^0_{0.99}$	32.1±0.0	45.2±0.1	39.2±0.6	43.1±0.3	59.9±0.0	11.4±0.0	38.5
$\text{ECDG}^1_{0.0}$	29.0±0.1	36.5±0.3	29.1±0.2	35.5±0.1	57.2±0.0	8.0±0.0	32.5
$\text{ECDG}^1_{0.2}$	30.7±0.0	40.0±0.2	29.3±0.1	35.2±0.1	57.5±0.0	8.0±0.0	33.5
$\text{ECDG}^1_{0.4}$	31.3±0.2	39.6±0.4	28.2±0.1	35.2±0.0	57.2±0.0	7.5±0.0	33.2
$\text{ECDG}^1_{0.6}$	29.6±0.0	37.5±0.1	26.5±0.2	35.9±0.2	57.3±0.0	6.9±0.0	32.3
$\text{ECDG}^1_{0.8}$	27.0±0.0	36.9±0.1	20.9±0.0	35.4±0.2	55.6±0.0	6.8±0.0	30.4
$\text{ECDG}^1_{0.99}$	26.1±0.0	36.9±0.1	20.9±0.0	36.0±0.2	55.7±0.0	6.7±0.0	30.4
$\text{ECDG}^2_{0.0}$	27.3±0.0	32.9±0.1	13.8±0.0	35.1±0.1	53.3±0.0	6.4±0.0	28.1
$\text{ECDG}^2_{0.2}$	27.3±0.0	34.6±0.1	14.2±0.0	35.3±0.0	53.1±0.0	6.4±0.0	28.5
$\text{ECDG}^2_{0.4}$	28.0±0.0	32.8±0.1	12.9±0.0	35.3±0.1	53.3±0.0	5.8±0.0	28.0
$\text{ECDG}^2_{0.6}$	27.1±0.1	30.9±0.0	11.7±0.1	36.0±0.1	53.0±0.0	5.8±0.0	27.4
$\text{ECDG}^2_{0.8}$	23.3±0.0	30.0±0.1	6.8±0.0	36.1±0.2	52.3±0.0	6.0±0.0	25.8
$\text{ECDG}^2_{0.99}$	22.6±0.0	29.4±0.1	6.3±0.0	35.9±0.2	52.3±0.0	6.0±0.0	25.4
$\text{ECDG}^3_{0.0}$	24.8±0.2	29.6±0.1	9.5±0.0	37.9±0.2	51.7±0.0	5.2±0.0	26.4
$\text{ECDG}^3_{0.2}$	26.1±0.0	30.8±0.0	9.3±0.0	36.3±0.2	51.3±0.0	5.2±0.0	26.5
$\text{ECDG}^3_{0.4}$	25.5±0.0	27.5±0.0	8.6±0.0	36.9±0.2	51.3±0.0	4.9±0.0	25.8
$\text{ECDG}^3_{0.6}$	24.4±0.0	27.2±0.0	7.6±0.0	36.3±0.2	50.7±0.0	4.9±0.0	25.2
$\text{ECDG}^3_{0.8}$	21.7±0.0	26.4±0.0	4.0±0.0	36.2±0.2	50.3±0.0	5.1±0.0	23.9
$\text{ECDG}^3_{0.99}$	20.9±0.0	25.5±0.0	3.5±0.0	36.2±0.2	50.3±0.0	5.1±0.0	23.6
Default	16.3±0.0	15.7±0.0	1.8±0.0	36.6±0.0	50.3±0.0	4.1±0.0	20.8

Classifying the quality of these first results for structural graphs, this paragraph compares them in a brief excursus to other similar studies that also use the *MoJoFM* metric or at least the predecessor metric *MoJo* (as shown in [70], *MoJo* produces just slightly higher values than *MoJoFM*). The work that introduces *MoJoFM* [70] performs a clustering experiment with several clustering algorithms on two sample projects, *Linux* and *TOBEY*. The experiment yields *MoJoFM* values between 37 and 75 for *Linux*, and between 24 and 66 for *TOBEY*; their results for the *Bunch* algorithm (used in a similar setup as in the present work) are 74 for *Linux* and 58 for *TOBEY*. Results from other studies are similar: For example, Maqbool and Babri [46] obtained values from 30 to 62 with the *MoJo* metric (averaged over several expert decompositions). Compared to these related studies, the present results are competitive, that is, the clustering process seems to yield good clustering results in the current setup. A stronger statement cannot be made because the *MoJoFM* values are not directly comparable for different projects as previously discussed.

3.2.2 Results for Evolutionary Graphs

The $\mathrm{ECDG}^{0}_{0.4}$, which has a support threshold of 0 and a confidence threshold of 0.4, delivers the best average value over all projects for evolutionary dependencies (40.2), closely followed by the other evolutionary graphs with a support threshold of 0. This clustering quality is nearly equal to the values of the CIG (41.9) and the CAG (37.9) but clearly lower than the CUG (53.2) and SCDG (54.3) values. Thus, in the average case (at least for the current selection of sample projects) the evolutionary data source performs just as well as the inheritance or the aggregation dependency information.

Observing the changes of the average clustering quality for different support filters, the clustering quality steadily falls with a higher support threshold. Either the filtering does not work as expected or the effect of decreasing the number of dependencies is stronger than the effect of enhancing the dependency reliability. In contrast, increasing the confidence threshold slightly improves the clustering quality for lower threshold values although the filter is also reducing the dependency information. But after reaching the maximum at a confidence threshold of 0.2 or 0.4 the quality decreases again. This effect is a clear evidence that filtering by confidence is able to improve the dependency reliability.

These characteristics of the average value need not be valid for every project. For instance, the evolutionary clustering quality for the *Tomcat* project is very low (4.9 - 11.4) and far from being competitive to any structural dependency graph. As already noticed before, this effect probably results from the relatively sparse evolutionary class dependency graph. In contrast, for *JUnit* the evolutionary dependencies yield better results than any structural dependency kind (structural: 58.8; evolutionary: 61.9), and for *JFreeChart* the relation is nearly balanced (structural: 44.2; evolutionary: 43.2). The effect of the filter is very similar for all projects. The *JFtp* project, for example, is a marginal exception with its best evolutionary clustering quality at the $\mathrm{ECDG}^{0}_{0.99}$ (43.1).

3.2.3 Experimental Quality and Impact

The default decomposition qualities surprisingly cover a very wide range of *MoJoFM* values, from 1.8 (*JFreeChart*) up to 50.3 (*JUnit*). This observation clearly underlines 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 the best clustering and the default clustering is also heavily varying for the different projects: It ranges from 11.6 (*JUnit*) up to 49.8 (*JEdit*) and 50.4 (*Tomcat*). This suggests that the clustering results for *JEdit* and *Tomcat* are better than the results of *JUnit* although their absolute *MoJoFM* values are similar (JEdit: 65.5; JUnit: 58.8; Tomcat: 54.4).

The precision of the measured *MoJoFM* quality values is yet not discussed. Mitchell and Mancoridis [50] showed that the results of *Bunch* either cluster closely around a single constant *TurboMQ* value (the internally used clustering quality metric of *Bunch*) or around two *TurboMQ* values. The algorithm just has one or two stable decomposition states. Probably, this has a similar impact on the *MoJoFM* distribution. The results in Table 3.2 confirm this assumption because most results can be reproduced precisely in repeated runs (low standard errors of 0.0–0.2), while few other results have a much worse precision (0.5–1.5), which is probably caused by several peaks in the quality distribution. The three most imprecise mean values are the clustering qualities for *JEdit* CUG (1.7), *JEdit* SCDG (1.6), and *JFreeChart* $\mathrm{ECDG}^{0}_{0.2}$ (1.0). They belong to *JEdit* and *JFreeChart*, the two largest sample projects by number of classes. It underlines that the *Bunch* is supposed to work more and more imprecise with an increasing graph size because of a heavily extended search space of the clustering heuristic. The clustering results, however, are precise enough to interpret *MoJoFM* differences of 1 or more as real differences, at least for all average values and for most project specific values.

Since this first experiment addresses the clustering quality based on single data sources, it already confirms for many cases that *Software evolution provides valuable information for software clustering* (Hypothesis 1) because competitive clustering results are reached by exclusively using evolutionary dependencies. In general, the results of the best evolutionary graph is comparable to the graphs based on inheritance or aggregation. When the evolutionary information is sparse, however, a meaningful evolutionary clustering result cannot be created. But that does not automatically mean that the information is not valuable—it can be used in other ways as the following experiments show.

3.3 Analyzing the Dependency Quality

The idea behind the concept of support and confidence of evolutionary dependencies is that the higher the support or confidence value the stronger the dependency. One can assume that this statement also holds for the present software clustering application: The evolutionary dependencies get more and more reliable for increasing support and confidence thresholds. The results of the previous experiment, however, cannot directly support this assumption because two contrary effects might collide:

- As discussed, stronger filter settings might provide data with higher quality.
- Concurrently, stronger filter settings reduce the available data.

To examine the influence of both effects, this section conducts additional analyses. The results will also help to identify promising configurations for a combination of different data sources.

3.3.1 Package Intra-Edges

In the dependency graph all edges that connect classes of the same package are called *intra-edges*. They enable the clustering algorithm to detect a similar package structure. All edges that connect classes from different packages are called *inter-edges*. In contrast, they influence the clustering result negatively. Hence, the percentage of intra-edges among all edges of the graph provides a measure for the dependency quality. It is independent of the total amount of available dependencies. Table 3.3 lists the values of this measure (first value) for the same selection of graphs as in the previous clustering experiment. Additionally, the amount of available data is expressed as the percentage of nodes with at least one in- or out-going edge (second value, printed smaller); this value is called *node coverage*.

The average values in the last column of the table show similar intra-edge ratios for the structural graphs, ranging from 39% to 49%, and more varying ratios for evolutionary

TABLE 3.3: Percentage of package intra-edges (printed normal) and node coverage (printed small) for single structural and evolutionary dependency graphs.

	Azureus	*JEdit*	*JFreeChart*	*JFtp*	*JUnit*	*Tomcat*	Average
CIG	25%$_{60\%}$	70%$_{44\%}$	69%$_{50\%}$	40%$_{56\%}$	26%$_{42\%}$	47%$_{51\%}$	46%$_{51\%}$
CAG	42%$_{52\%}$	81%$_{64\%}$	41%$_{29\%}$	26%$_{59\%}$	56%$_{21\%}$	46%$_{61\%}$	49%$_{48\%}$
CUG	32%$_{98\%}$	59%$_{99\%}$	27%$_{97\%}$	74%$_{42\%}$	37%$_{91\%}$	35%$_{94\%}$	44%$_{87\%}$
SCDG	31%$_{99\%}$	59%$_{100\%}$	29%$_{98\%}$	39%$_{81\%}$	37%$_{91\%}$	35%$_{94\%}$	39%$_{94\%}$
$\text{ECDG}^0_{0.0}$	16%$_{54\%}$	23%$_{65\%}$	40%$_{66\%}$	37%$_{68\%}$	29%$_{35\%}$	30%$_{15\%}$	29%$_{50\%}$
$\text{ECDG}^0_{0.2}$	32%$_{54\%}$	36%$_{64\%}$	47%$_{66\%}$	39%$_{68\%}$	31%$_{35\%}$	31%$_{15\%}$	36%$_{50\%}$
$\text{ECDG}^0_{0.4}$	39%$_{51\%}$	43%$_{64\%}$	54%$_{64\%}$	41%$_{65\%}$	35%$_{35\%}$	35%$_{14\%}$	41%$_{49\%}$
$\text{ECDG}^0_{0.6}$	45%$_{45\%}$	51%$_{62\%}$	67%$_{57\%}$	43%$_{64\%}$	41%$_{33\%}$	37%$_{14\%}$	47%$_{46\%}$
$\text{ECDG}^0_{0.8}$	47%$_{39\%}$	53%$_{60\%}$	63%$_{56\%}$	44%$_{64\%}$	42%$_{32\%}$	37%$_{14\%}$	48%$_{44\%}$
$\text{ECDG}^0_{0.99}$	48%$_{39\%}$	52%$_{60\%}$	63%$_{56\%}$	44%$_{64\%}$	42%$_{32\%}$	37%$_{14\%}$	48%$_{44\%}$
$\text{ECDG}^1_{0.0}$	19%$_{40\%}$	29%$_{49\%}$	78%$_{36\%}$	41%$_{49\%}$	23%$_{15\%}$	48%$_{06\%}$	40%$_{33\%}$
$\text{ECDG}^1_{0.2}$	36%$_{39\%}$	43%$_{49\%}$	83%$_{36\%}$	45%$_{49\%}$	24%$_{15\%}$	46%$_{06\%}$	46%$_{33\%}$
$\text{ECDG}^1_{0.4}$	45%$_{34\%}$	54%$_{48\%}$	89%$_{34\%}$	40%$_{49\%}$	25%$_{15\%}$	49%$_{06\%}$	50%$_{31\%}$
$\text{ECDG}^1_{0.6}$	45%$_{30\%}$	65%$_{45\%}$	94%$_{32\%}$	43%$_{49\%}$	21%$_{13\%}$	56%$_{04\%}$	54%$_{29\%}$
$\text{ECDG}^1_{0.8}$	51%$_{21\%}$	83%$_{41\%}$	99%$_{22\%}$	46%$_{42\%}$	19%$_{09\%}$	71%$_{03\%}$	61%$_{23\%}$
$\text{ECDG}^1_{0.99}$	54%$_{19\%}$	84%$_{40\%}$	99%$_{22\%}$	46%$_{42\%}$	19%$_{09\%}$	71%$_{03\%}$	62%$_{23\%}$
$\text{ECDG}^2_{0.0}$	20%$_{32\%}$	33%$_{39\%}$	85%$_{16\%}$	37%$_{42\%}$	18%$_{07\%}$	67%$_{03\%}$	43%$_{23\%}$
$\text{ECDG}^2_{0.2}$	36%$_{31\%}$	47%$_{39\%}$	87%$_{15\%}$	40%$_{42\%}$	19%$_{07\%}$	65%$_{03\%}$	49%$_{23\%}$
$\text{ECDG}^2_{0.4}$	44%$_{27\%}$	59%$_{38\%}$	87%$_{14\%}$	37%$_{42\%}$	18%$_{07\%}$	61%$_{02\%}$	51%$_{22\%}$
$\text{ECDG}^2_{0.6}$	45%$_{21\%}$	76%$_{34\%}$	98%$_{12\%}$	39%$_{42\%}$	20%$_{06\%}$	69%$_{02\%}$	58%$_{20\%}$
$\text{ECDG}^2_{0.8}$	49%$_{13\%}$	90%$_{31\%}$	100%$_{05\%}$	40%$_{35\%}$	20%$_{03\%}$	82%$_{02\%}$	63%$_{15\%}$
$\text{ECDG}^2_{0.99}$	56%$_{11\%}$	93%$_{29\%}$	100%$_{05\%}$	38%$_{35\%}$	20%$_{03\%}$	82%$_{02\%}$	65%$_{14\%}$
$\text{ECDG}^3_{0.0}$	20%$_{25\%}$	34%$_{30\%}$	92%$_{09\%}$	40%$_{36\%}$	23%$_{03\%}$	60%$_{02\%}$	45%$_{17\%}$
$\text{ECDG}^3_{0.2}$	35%$_{24\%}$	48%$_{30\%}$	92%$_{09\%}$	42%$_{36\%}$	21%$_{03\%}$	60%$_{02\%}$	50%$_{17\%}$
$\text{ECDG}^3_{0.4}$	43%$_{20\%}$	61%$_{29\%}$	88%$_{08\%}$	39%$_{36\%}$	17%$_{03\%}$	50%$_{01\%}$	50%$_{16\%}$
$\text{ECDG}^3_{0.6}$	44%$_{15\%}$	76%$_{26\%}$	99%$_{06\%}$	42%$_{36\%}$	00%$_{01\%}$	40%$_{01\%}$	50%$_{14\%}$
$\text{ECDG}^3_{0.8}$	54%$_{10\%}$	90%$_{23\%}$	100%$_{02\%}$	38%$_{29\%}$	–	67%$_{01\%}$	–
$\text{ECDG}^3_{0.99}$	68%$_{08\%}$	95%$_{21\%}$	100%$_{02\%}$	34%$_{26\%}$	–	67%$_{01\%}$	–

graphs, ranging from 29% to 65%. Thus, with the right filter setting, the evolutionary dependencies provide better data qualities in terms of the intra-edge ratios. As seen in the first experiment, this does not lead to better clustering results because the evolutionary dependency data is too sparse. This is especially the case for the high quality filter settings with, for example, 65% of intra-edges and only 14% of covered nodes ($\text{ECDG}^2_{0.99}$) or 62% of intra-edges and only 23% of covered nodes ($\text{ECDG}^1_{0.99}$). But also for the unfiltered $\text{ECDG}^0_{0.0}$ only 50% of all nodes are covered by evolutionary dependencies, i.e., at least 50% of nodes cannot be reasonably clustered because of missing information. In contrast, the SCDG, which provides the overall best clustering results in the first experiment, has an average node coverage rate of 94% while the intra-edge ratio is low (only 39%).

The intra-edge percentage values show that the data quality clearly increases with higher confidence values. In contrast, varying the support, just the first step (incrementing the support threshold from 0 to 1) enhances the quality. Further increments show no relevant effect. In both cases of increasing the thresholds, however, the node coverage rate drops step by step as expected.

These characteristics of the average values can be observed for the *Azureus*, *JEdit*, *JFreeChart*, and *Tomcat* project in a similar way. *JFtp* and *JUnit*, though, show different behavior for the filtered evolutionary data:

- For *JFtp* the filtering nearly has no impact on the intra-edge data quality, which is only varying in a narrow range of 34% and 46% and does not increase generally with stronger filter settings.
- The intra-edge data quality for *JUnit* even shows inverse effects compared to the average values: It decreases for higher support threshold and changes diversely for higher confidence values.

3.3.2 Incomplete *MoJoFM* Comparison

The percentage of intra-edges is not a direct measure of the data quality for the application of software clustering because it is actually applied before the clustering. To measure the quality directly but independently from the data density, the clustering results of the first experiment can be reused. But they are assessed slightly different with the help of the *MoJoFM* metric: Instead of using decompositions of all classes, the decompositions are restricted to only those classes that are covered by dependencies of the current dependency graph. Hence, every clustering result is compared to an individual subset of the reference decomposition that exactly consists of the set of covered classes. It is, however, a limiting factor of such an evaluation that the resulting *MoJoFM* values are not comparable because they are normalized with different reference decompositions. But as the reference decomposition for two evolutionary graphs with similar threshold values only differs marginally, this effect can be considered as weak for such two graphs. Nevertheless, the results must be interpreted more cautiously.

Table 3.4 provides the *MoJoFM* values of this incomplete comparison. The results of this direct data quality measurement are similar to those of the previous indirect intra-edge measurement: The data quality in *MoJoFM* points is ranging up to 57.1 (SCDG) for structural dependencies while higher values can be reached with strongly filtered evolutionary data (up to 65.8). Increasing the confidence threshold, the data quality increases. As in the indirect measurement, a higher support value improves the quality only for switching from 0 to 1 significantly. Also *JFtp* and *JUnit* are the exceptions of these trends: The *MoJoFM* quality of *JFtp* decompositions is nearly constant, and the filtering for the *JUnit* project results in diverse behavior.

TABLE 3.4: *MoJoFM* clustering quality with incomplete reference decompositions based on single structural and evolutionary dependency graphs.

	Azureus	*JEdit*	*JFreeChart*	*JFtp*	*JUnit*	*Tomcat*	Average
CIG	35.3$\pm$0.3	65.9$\pm$1.1	66.2$\pm$0.5	76.0$\pm$0.8	43.5$\pm$0.1	55.2$\pm$0.2	57.0
CAG	43.9$\pm$0.6	63.3$\pm$1.2	44.4$\pm$0.2	64.4$\pm$0.5	43.4$\pm$0.1	53.1$\pm$0.0	52.1
CUG	53.4$\pm$0.4	63.7$\pm$1.7	41.3$\pm$0.6	64.4$\pm$0.1	62.6$\pm$0.2	57.1$\pm$0.5	57.1
SCDG	50.4$\pm$1.0	65.6$\pm$1.6	44.1$\pm$0.4	63.8$\pm$0.3	62.2$\pm$0.2	56.3$\pm$0.6	57.1
$\text{ECDG}^0_{0.0}$	44.9$\pm$0.2	46.8$\pm$0.2	56.8$\pm$0.3	50.0$\pm$0.3	49.4$\pm$0.2	48.0$\pm$0.2	49.3
$\text{ECDG}^0_{0.2}$	50.2$\pm$0.7	56.1$\pm$0.6	57.8$\pm$1.6	47.7$\pm$0.4	47.6$\pm$0.1	48.5$\pm$0.1	51.3
$\text{ECDG}^0_{0.4}$	51.9$\pm$0.1	57.5$\pm$0.3	62.1$\pm$0.1	51.2$\pm$0.7	53.3$\pm$0.2	52.0$\pm$0.2	54.7
$\text{ECDG}^0_{0.6}$	52.9$\pm$0.4	60.3$\pm$0.1	65.4$\pm$0.1	52.5$\pm$0.5	55.4$\pm$0.1	54.4$\pm$0.1	56.8
$\text{ECDG}^0_{0.8}$	56.6$\pm$0.9	59.7$\pm$0.2	65.8$\pm$0.1	53.0$\pm$0.5	52.4$\pm$0.1	54.4$\pm$0.1	57.0
$\text{ECDG}^0_{0.99}$	55.9$\pm$0.1	58.8$\pm$0.1	64.6$\pm$1.1	53.6$\pm$0.4	52.2$\pm$0.1	54.3$\pm$0.0	56.6
$\text{ECDG}^1_{0.0}$	47.1$\pm$0.1	52.4$\pm$0.6	71.5$\pm$0.5	47.6$\pm$0.2	49.1$\pm$0.2	60.5$\pm$0.4	54.7
$\text{ECDG}^1_{0.2}$	50.8$\pm$0.1	59.6$\pm$0.4	72.2$\pm$0.2	46.9$\pm$0.2	51.8$\pm$0.2	62.1$\pm$0.0	57.2
$\text{ECDG}^1_{0.4}$	54.1$\pm$0.5	60.1$\pm$0.8	73.7$\pm$0.3	46.9$\pm$0.1	53.1$\pm$0.2	55.6$\pm$0.0	57.2
$\text{ECDG}^1_{0.6}$	52.6$\pm$0.1	60.2$\pm$0.2	74.6$\pm$0.6	48.4$\pm$0.4	52.9$\pm$0.1	57.3$\pm$0.1	57.7
$\text{ECDG}^1_{0.8}$	56.2$\pm$0.1	64.3$\pm$0.3	85.0$\pm$0.1	53.9$\pm$0.5	56.2$\pm$0.1	67.1$\pm$0.2	63.8
$\text{ECDG}^1_{0.99}$	56.1$\pm$0.1	65.7$\pm$0.3	85.2$\pm$0.2	55.6$\pm$0.4	56.4$\pm$0.2	66.8$\pm$0.1	64.3
$\text{ECDG}^2_{0.0}$	50.3$\pm$0.1	56.7$\pm$0.3	70.6$\pm$0.3	46.1$\pm$0.3	33.3$\pm$0.0	71.4$\pm$0.0	54.8
$\text{ECDG}^2_{0.2}$	50.1$\pm$0.1	61.3$\pm$0.2	75.1$\pm$0.3	46.6$\pm$0.1	29.8$\pm$0.4	71.4$\pm$0.0	55.7
$\text{ECDG}^2_{0.4}$	54.6$\pm$0.1	57.7$\pm$0.3	71.7$\pm$0.2	46.7$\pm$0.2	33.3$\pm$0.0	63.6$\pm$0.0	54.6
$\text{ECDG}^2_{0.6}$	59.2$\pm$0.4	59.5$\pm$0.1	77.5$\pm$0.5	48.4$\pm$0.4	37.5$\pm$0.0	70.0$\pm$0.0	58.7
$\text{ECDG}^2_{0.8}$	56.6$\pm$0.0	63.7$\pm$0.2	91.7$\pm$0.1	52.7$\pm$0.5	50.0$\pm$0.0	77.8$\pm$0.0	65.4
$\text{ECDG}^2_{0.99}$	60.0$\pm$0.0	64.9$\pm$0.2	89.8$\pm$0.2	52.1$\pm$0.6	50.0$\pm$0.0	77.8$\pm$0.0	65.8
$\text{ECDG}^3_{0.0}$	45.8$\pm$0.8	61.1$\pm$0.3	79.3$\pm$0.3	53.9$\pm$0.6	42.9$\pm$0.0	71.4$\pm$0.0	59.1
$\text{ECDG}^3_{0.2}$	51.4$\pm$0.1	65.3$\pm$0.2	77.3$\pm$0.3	49.1$\pm$0.6	28.6$\pm$0.0	71.4$\pm$0.0	57.2
$\text{ECDG}^3_{0.4}$	52.5$\pm$0.1	54.9$\pm$0.1	81.1$\pm$0.3	50.9$\pm$0.5	33.3$\pm$0.0	60.0$\pm$0.0	55.5
$\text{ECDG}^3_{0.6}$	56.3$\pm$0.1	60.0$\pm$0.0	87.2$\pm$0.3	48.9$\pm$0.5	33.3$\pm$0.0	60.0$\pm$0.0	57.6
$\text{ECDG}^3_{0.8}$	55.6$\pm$0.1	63.0$\pm$0.1	98.7$\pm$0.4	50.9$\pm$0.7	–	75.0$\pm$0.0	–
$\text{ECDG}^3_{0.99}$	62.5$\pm$0.0	63.9$\pm$0.1	96.0$\pm$0.8	42.0$\pm$0.8	–	75.0$\pm$0.0	–

Finally, the results of the indirect as well as of the direct evaluation indicate that the filtering of evolutionary graphs by support and confidence works in practice as derived from theory for all projects except *JFtp* and *JUnit*. Choosing a higher support threshold than 1, however, is not recommendable because it would just restrict the data without increasing the quality significantly. Moreover, the data quality tends to be the strength of the evolutionary graphs while the data density tends to be the strength of the structural graphs.

3.4 Clustering with Combined Data Sources

While Hypothesis 1 deals with the autonomous clustering quality of evolutionary dependencies, Hypothesis 2 focuses on combined dependencies from structural and evolutionary data sources. Filtering evolutionary data was only partly successful because it decreases the data density more than it increases the data quality—the slightly filtered $\mathrm{ECDG}^{0}_{0.4}$ produces the best results. As the data quality of the evolutionary dependencies is good, yet sparse, it can be presumed that integrating the evolutionary dependencies into the dense structural data, which has a lower dependency quality, improves the overall clustering results. The union operations on graphs, defined in Section 2.3.3, provide tools to realize the necessary data integration.

Combining each of the four structural graphs with each of the 24 evolutionary graphs used in the first experiment would result in too high clustering costs. Hence, the set of evolutionary graphs has to be restricted to a smaller but still representative selection. The following selection is motivated by the analysis of the data quality in Section 3.3.

- $\mathrm{ECDG}^{0}_{0.0}$ (intra-edge ratio: 30%; incomplete *MoJoFM*: 49.2; node coverage: 51%): This graph contains all evolutionary dependencies without any filtering (i.e., the raw data) and thus provides the most complete evolutionary data set.
- $\mathrm{ECDG}^{0}_{0.4}$ (intra-edge ratio: 41%; incomplete *MoJoFM*: 54.4; node coverage: 50%): The evolutionary graph that performs best in the first experiment is obviously a candidate for further high quality clustering results.
- $\mathrm{ECDG}^{0}_{0.8}$ (intra-edge ratio: 47%; incomplete *MoJoFM*: 56.8; node coverage: 45%): A higher confidence threshold increases the data quality while it decreases the data density only moderately.
- $\mathrm{ECDG}^{1}_{0.4}$ (intra-edge ratio: 51%; incomplete *MoJoFM*: 57.3; node coverage: 31%): Incrementing the support threshold from 0 to 1 has a similar effect (according to Section 3.3 further increments are inefficient).
- $\mathrm{ECDG}^{1}_{0.8}$ (intra-edge ratio: 63%; incomplete *MoJoFM*: 62.3; node coverage: 23%): Among the evolutionary graphs with the highest data quality, this graph is the one with the best dependency density.

3.4.1 Simple Union

The non-weighted union operation (Definition 2.10) is a simple method to combine two graphs: It just performs a normal set union operation each on the two sets of nodes and the two sets of edges. Thus, all dependencies are preserved uniting a structural with an evolutionary dependency graph. The information which dependencies were contained in which of the original graphs is lost however.

Table 3.5 presents the *MoJoFM* quality of different clustering decompositions (again compared as complete decompositions): It contrasts the single structural or evolutionary data sources to the combined data sources. The latter ones are created by pairwise combinations of the structural and evolutionary graphs.

The graph combination with the simple union operation clearly improves the clustering quality in nearly all cases comparing the combined graph to the two contained original graphs. From the perspective of the structural graphs, the average clustering quality rises

- for the CIG from 41.9 to 48.5 ($\mathrm{CIG} \cup \mathrm{ECDG}^0_{0.4}$),
- for the CAG from 37.9 to 46.9 ($\mathrm{CAG} \cup \mathrm{ECDG}^0_{0.8}$),
- for the CUG from 53.2 to 56.5 ($\mathrm{CUG} \cup \mathrm{ECDG}^0_{0.8}$),
- and, most important, for the SCDG from 54.3 to 57.4 ($\mathrm{SCDG} \cup \mathrm{ECDG}^0_{0.8}$).

Furthermore, the clustering quality is not only improved in these best case combinations but also in nearly all other cases, except for the union of the CUG and SCDG with the $\mathrm{ECDG}^0_{0.0}$. Switching the perspective to the evolutionary graphs, even every combination is an improvement of the original clustering with exclusively evolutionary dependencies. The combinations with the SCDG provide the best results.

On project level, the effect of improving a structural graph with at least one evolutionary graph sustains unexceptional for each project, even slightly in the *Tomcat* project with its very sparse evolutionary dependencies and in the *JUnit* project with its partly inverse filtering behavior. And vice versa, improving an evolutionary graph, the effect holds also for every project while it is, however, very small for the *JUnit* project, which has a slightly better evolutionary than structural clustering quality.

Finally, this second clustering experiment demonstrates that it is possible to increase the quality of a structural dependency based clustering by integrating evolutionary data, even for projects with only sparse evolutionary dependency information. Thus, the

Table 3.5: *MoJoFM* clustering quality based on combined structural and evolutionary dependency graphs using the simple union operation ∪. The best average results are highlighted gray for each block.

	Azureus	*JEdit*	*JFreeChart*	*JFtp*	*JUnit*	*Tomcat*	Average
CIG	30.2±0.2	42.0±0.5	35.4±0.3	53.4±0.4	58.0±0.1	32.8±0.1	41.9
CAG	28.6±0.3	47.1±0.7	14.3±0.0	45.7±0.3	55.9±0.0	35.8±0.0	37.9
CUG	52.4±0.4	63.5±1.7	40.6±0.6	49.4±0.0	58.8±0.2	54.5±0.5	53.2
SCDG	49.9±1.0	65.5±1.6	44.2±0.4	54.0±0.3	58.4±0.2	54.0±0.5	54.3
$\text{ECDG}^0_{0.0}$	31.7±0.1	40.4±0.1	40.7±0.2	41.5±0.2	60.5±0.1	11.1±0.0	37.7
$\text{ECDG}^0_{0.4}$	35.6±0.1	47.0±0.2	43.2±0.1	42.3±0.4	61.9±0.1	11.3±0.0	40.2
$\text{ECDG}^0_{0.8}$	32.6±0.3	45.9±0.1	39.8±0.1	42.7±0.3	60.0±0.0	11.4±0.0	38.7
$\text{ECDG}^1_{0.4}$	31.3±0.2	39.6±0.4	28.2±0.1	35.2±0.0	57.2±0.0	7.5±0.0	33.2
$\text{ECDG}^1_{0.8}$	27.0±0.0	36.9±0.1	20.9±0.0	35.4±0.2	55.6±0.0	6.8±0.0	30.4
$\text{CIG} \cup \text{ECDG}^0_{0.0}$	33.9±0.2	44.1±0.9	49.8±0.1	45.0±0.2	58.8±0.1	34.4±0.4	44.3
$\text{CIG} \cup \text{ECDG}^0_{0.4}$	38.5±0.4	53.1±0.3	53.8±0.4	51.0±0.3	59.8±0.0	34.6±0.1	48.5
$\text{CIG} \cup \text{ECDG}^0_{0.8}$	36.4±0.1	52.7±0.9	55.1±0.2	47.3±0.3	60.6±0.0	34.6±0.0	47.8
$\text{CIG} \cup \text{ECDG}^1_{0.4}$	35.7±0.2	52.8±0.2	46.5±0.7	53.3±0.3	59.5±0.0	34.1±0.0	47.0
$\text{CIG} \cup \text{ECDG}^1_{0.8}$	34.0±0.1	54.4±0.3	44.4±0.6	57.2±0.2	59.6±0.0	34.1±0.0	47.3
$\text{CAG} \cup \text{ECDG}^0_{0.0}$	33.3±0.1	47.5±0.4	43.3±0.1	44.3±0.1	60.8±0.1	36.1±0.1	44.2
$\text{CAG} \cup \text{ECDG}^0_{0.4}$	35.8±0.1	56.7±0.7	44.6±0.1	45.7±0.3	62.0±0.1	36.0±0.0	46.8
$\text{CAG} \cup \text{ECDG}^0_{0.8}$	34.4±0.1	58.0±1.0	41.3±0.1	50.2±0.6	61.5±0.0	35.8±0.0	46.9
$\text{CAG} \cup \text{ECDG}^1_{0.4}$	33.3±0.0	55.5±0.4	31.9±0.5	45.1±0.1	57.9±0.0	35.7±0.4	43.2
$\text{CAG} \cup \text{ECDG}^1_{0.8}$	32.7±0.0	54.3±1.1	27.1±0.1	50.2±0.3	57.3±0.0	35.9±0.0	42.9
$\text{CUG} \cup \text{ECDG}^0_{0.0}$	49.2±0.5	53.6±1.2	50.4±0.2	46.1±0.3	56.5±0.3	51.0±1.2	51.1
$\text{CUG} \cup \text{ECDG}^0_{0.4}$	52.2±1.0	69.0±0.1	54.3±0.3	49.2±0.4	59.2±0.1	52.8±1.4	56.1
$\text{CUG} \cup \text{ECDG}^0_{0.8}$	53.1±0.7	65.2±1.5	52.1±0.4	54.6±0.4	58.6±0.1	55.6±0.5	56.5
$\text{CUG} \cup \text{ECDG}^1_{0.4}$	53.0±0.7	65.9±1.7	52.9±0.5	46.1±0.3	58.1±0.2	55.3±1.3	55.2
$\text{CUG} \cup \text{ECDG}^1_{0.8}$	52.3±1.1	65.6±1.5	47.9±0.7	48.2±0.4	59.1±0.1	56.5±0.7	54.9
$\text{SCDG} \cup \text{ECDG}^0_{0.0}$	48.4±0.6	54.8±1.2	53.1±0.2	47.7±0.2	57.2±0.2	51.7±0.9	52.1
$\text{SCDG} \cup \text{ECDG}^0_{0.4}$	50.6±0.8	63.7±2.0	55.9±0.2	52.4±0.4	59.1±0.1	56.1±0.2	56.3
$\text{SCDG} \cup \text{ECDG}^0_{0.8}$	52.7±0.1	68.7±0.7	53.3±1.1	55.1±0.4	58.8±0.1	55.9±0.2	57.4
$\text{SCDG} \cup \text{ECDG}^1_{0.4}$	52.3±0.4	64.6±2.1	56.0±0.5	54.0±0.4	58.9±0.1	57.0±0.2	57.1
$\text{SCDG} \cup \text{ECDG}^1_{0.8}$	52.6±0.2	64.1±2.1	50.5±0.8	61.0±0.2	58.3±0.2	56.6±0.2	57.2

experiment clearly confirms Hypothesis 2 for the current use case of architecture recovery. Moreover, it extends the validity of Hypothesis 1 to all projects: Even very sparse evolutionary information is valuable for software clustering because it can be used to improve the structural clustering result.

3.4.2 Weighted Union

A certain structural dependency may exist because a developer has misplaced a method. Similarly, a certain 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. Thus, dependencies that are included in the structural as well as the evolutionary graph are considered more reliable than those that are only member of one of the graphs.

The weighted union operation extends the simple union operation by allowing different weights for dependencies that are either included in only the first, in only the second, or in both original graphs—a weighted union with weights of 1 for all three groups is equivalent to the simple union. These weights influence the *Bunch* clustering algorithm, or more exactly, its internal quality metric *ITurboMQ* (Section 2.4.3). It might be possible to improve the clustering results further by choosing other weights, thus, to strengthen the evidence of the previous experiment. Deduced from the considerations above, it is reasonable to choose a higher weight for the twice occurring dependencies and a lower weight for the single occurring dependencies: The weight of the exclusively structural dependencies is set to 1 as well as the weight of the exclusively evolutionary dependencies. To stress the importance of the concurrently structural and evolutionary dependencies, their weight is set to 4 despite a naive choice for these twice occurring dependencies would be a weight of 2. Table 3.6 presents the results of a repeated union clustering experiment using the weighted union operation $\cup_{[1,4,1]}$ instead of the simple union operation $\cup$.

The clustering qualities for the weighted combinations are very similar to the ones obtained for the non-weighted combinations. A slight quality enhancement, however, can be observed in most cases: The best combined quality rises

- for the CAG from 46.9 ($\mathrm{CAG} \cup \mathrm{ECDG}^{0}_{0.8}$) to 47.4 ($\mathrm{CAG} \cup_{[1,4,1]} \mathrm{ECDG}^{0}_{0.8}$),
- for the CUG from 56.5 ($\mathrm{CUG} \cup \mathrm{ECDG}^{0}_{0.8}$) to 57.2 ($\mathrm{CUG} \cup_{[1,4,1]} \mathrm{ECDG}^{0}_{0.4}$),
- for the SCDG from 57.4 ($\mathrm{SCDG} \cup \mathrm{ECDG}^{0}_{0.8}$) to 58.5 ($\mathrm{SCDG} \cup_{[1,4,1]} \mathrm{ECDG}^{1}_{0.4}$).

TABLE 3.6: *MoJoFM* clustering quality based on combined structural and evolutionary dependency graphs using the weighted union operation $\cup_{[1,4,1]}$. The best average results are highlighted gray for each block.

	Azureus	*JEdit*	*JFreeChart*	*JFtp*	*JUnit*	*Tomcat*	Average
CIG	30.2$_{\pm0.2}$	42.0$_{\pm0.5}$	35.4$_{\pm0.3}$	53.4$_{\pm0.4}$	58.0$_{\pm0.1}$	32.8$_{\pm0.1}$	41.9
CAG	28.6$_{\pm0.3}$	47.1$_{\pm0.7}$	14.3$_{\pm0.0}$	45.7$_{\pm0.3}$	55.9$_{\pm0.0}$	35.8$_{\pm0.0}$	37.9
CUG	52.4$_{\pm0.4}$	63.5$_{\pm1.7}$	40.6$_{\pm0.6}$	49.4$_{\pm0.0}$	58.8$_{\pm0.2}$	54.5$_{\pm0.5}$	53.2
SCDG	49.9$_{\pm1.0}$	65.5$_{\pm1.6}$	44.2$_{\pm0.4}$	54.0$_{\pm0.3}$	58.4$_{\pm0.2}$	54.0$_{\pm0.5}$	54.3
ECDG$^0_{0.0}$	31.7$_{\pm0.1}$	40.4$_{\pm0.1}$	40.7$_{\pm0.2}$	41.5$_{\pm0.2}$	60.5$_{\pm0.1}$	11.1$_{\pm0.0}$	37.7
ECDG$^0_{0.4}$	35.6$_{\pm0.1}$	47.0$_{\pm0.2}$	43.2$_{\pm0.1}$	42.3$_{\pm0.4}$	61.9$_{\pm0.1}$	11.3$_{\pm0.0}$	40.2
ECDG$^0_{0.8}$	32.6$_{\pm0.3}$	45.9$_{\pm0.1}$	39.8$_{\pm0.1}$	42.7$_{\pm0.3}$	60.0$_{\pm0.0}$	11.4$_{\pm0.0}$	38.7
ECDG$^1_{0.4}$	31.3$_{\pm0.2}$	39.6$_{\pm0.4}$	28.2$_{\pm0.1}$	35.2$_{\pm0.0}$	57.2$_{\pm0.0}$	7.5$_{\pm0.0}$	33.2
ECDG$^1_{0.8}$	27.0$_{\pm0.0}$	36.9$_{\pm0.1}$	20.9$_{\pm0.0}$	35.4$_{\pm0.2}$	55.6$_{\pm0.0}$	6.8$_{\pm0.0}$	30.4
CIG $\cup_{[1,4,1]}$ ECDG$^0_{0.0}$	33.9$_{\pm0.3}$	44.3$_{\pm1.1}$	48.6$_{\pm1.2}$	44.8$_{\pm0.2}$	59.6$_{\pm0.0}$	34.3$_{\pm0.1}$	44.2
CIG $\cup_{[1,4,1]}$ ECDG$^0_{0.4}$	38.0$_{\pm0.2}$	51.7$_{\pm1.1}$	53.5$_{\pm0.6}$	51.4$_{\pm0.3}$	60.3$_{\pm0.0}$	34.8$_{\pm0.1}$	48.3
CIG $\cup_{[1,4,1]}$ ECDG$^0_{0.8}$	35.8$_{\pm0.2}$	53.4$_{\pm0.4}$	53.5$_{\pm1.0}$	47.0$_{\pm0.3}$	60.3$_{\pm0.0}$	34.0$_{\pm0.5}$	47.3
CIG $\cup_{[1,4,1]}$ ECDG$^1_{0.4}$	34.8$_{\pm0.3}$	52.9$_{\pm0.2}$	46.8$_{\pm0.6}$	54.0$_{\pm0.3}$	59.3$_{\pm0.1}$	34.3$_{\pm0.1}$	47.0
CIG $\cup_{[1,4,1]}$ ECDG$^1_{0.8}$	34.1$_{\pm0.1}$	54.3$_{\pm0.3}$	44.3$_{\pm0.3}$	56.9$_{\pm0.3}$	59.6$_{\pm0.0}$	34.0$_{\pm0.1}$	47.2
CAG $\cup_{[1,4,1]}$ ECDG$^0_{0.0}$	34.7$_{\pm0.6}$	50.9$_{\pm1.1}$	42.2$_{\pm0.6}$	45.7$_{\pm0.3}$	61.0$_{\pm0.1}$	36.6$_{\pm0.2}$	45.2
CAG $\cup_{[1,4,1]}$ ECDG$^0_{0.4}$	35.7$_{\pm0.5}$	59.1$_{\pm0.1}$	44.0$_{\pm0.5}$	45.5$_{\pm0.2}$	61.6$_{\pm0.1}$	36.3$_{\pm0.0}$	47.0
CAG $\cup_{[1,4,1]}$ ECDG$^0_{0.8}$	35.5$_{\pm0.0}$	59.6$_{\pm0.1}$	40.7$_{\pm0.7}$	51.3$_{\pm0.6}$	61.6$_{\pm0.0}$	36.0$_{\pm0.0}$	47.4
CAG $\cup_{[1,4,1]}$ ECDG$^1_{0.4}$	34.3$_{\pm0.0}$	56.1$_{\pm0.1}$	31.9$_{\pm0.1}$	44.9$_{\pm0.2}$	58.3$_{\pm0.0}$	36.1$_{\pm0.0}$	43.6
CAG $\cup_{[1,4,1]}$ ECDG$^1_{0.8}$	33.0$_{\pm0.1}$	56.3$_{\pm0.2}$	27.3$_{\pm0.1}$	47.5$_{\pm0.4}$	57.3$_{\pm0.0}$	36.0$_{\pm0.0}$	42.9
CUG $\cup_{[1,4,1]}$ ECDG$^0_{0.0}$	54.7$_{\pm1.0}$	60.9$_{\pm2.3}$	50.7$_{\pm1.1}$	48.3$_{\pm0.3}$	58.1$_{\pm0.1}$	51.9$_{\pm0.9}$	54.1
CUG $\cup_{[1,4,1]}$ ECDG$^0_{0.4}$	53.8$_{\pm0.7}$	69.5$_{\pm0.2}$	54.4$_{\pm0.1}$	50.6$_{\pm0.3}$	59.2$_{\pm0.2}$	55.4$_{\pm0.2}$	57.2
CUG $\cup_{[1,4,1]}$ ECDG$^0_{0.8}$	53.1$_{\pm0.6}$	68.6$_{\pm1.3}$	50.9$_{\pm1.2}$	55.8$_{\pm0.4}$	59.1$_{\pm0.1}$	53.9$_{\pm1.5}$	56.9
CUG $\cup_{[1,4,1]}$ ECDG$^1_{0.4}$	53.9$_{\pm0.6}$	69.5$_{\pm0.1}$	52.8$_{\pm0.9}$	47.7$_{\pm0.3}$	57.2$_{\pm0.1}$	56.1$_{\pm0.7}$	56.2
CUG $\cup_{[1,4,1]}$ ECDG$^1_{0.8}$	53.8$_{\pm0.1}$	69.7$_{\pm1.1}$	48.0$_{\pm0.5}$	46.4$_{\pm0.3}$	58.9$_{\pm0.2}$	57.0$_{\pm0.2}$	55.6
SCDG $\cup_{[1,4,1]}$ ECDG$^0_{0.0}$	53.6$_{\pm0.9}$	64.7$_{\pm1.5}$	52.1$_{\pm1.2}$	48.8$_{\pm0.3}$	58.7$_{\pm0.1}$	52.6$_{\pm0.7}$	55.1
SCDG $\cup_{[1,4,1]}$ ECDG$^0_{0.4}$	52.8$_{\pm0.9}$	67.9$_{\pm1.4}$	54.8$_{\pm0.9}$	55.4$_{\pm0.2}$	58.5$_{\pm0.2}$	55.1$_{\pm1.0}$	57.4
SCDG $\cup_{[1,4,1]}$ ECDG$^0_{0.8}$	52.7$_{\pm0.7}$	71.1$_{\pm0.9}$	54.4$_{\pm0.8}$	57.8$_{\pm0.7}$	58.7$_{\pm0.2}$	55.4$_{\pm0.4}$	58.4
SCDG $\cup_{[1,4,1]}$ ECDG$^1_{0.4}$	53.1$_{\pm0.8}$	69.0$_{\pm1.4}$	54.8$_{\pm0.5}$	62.1$_{\pm0.2}$	56.9$_{\pm0.2}$	55.3$_{\pm0.9}$	58.5
SCDG $\cup_{[1,4,1]}$ ECDG$^1_{0.8}$	52.8$_{\pm0.2}$	70.4$_{\pm0.6}$	50.0$_{\pm0.9}$	57.4$_{\pm0.5}$	58.0$_{\pm0.2}$	56.2$_{\pm0.2}$	57.5

Only for the CIG the best combined quality does not rise, it slightly decreases from 48.5 ($\text{CIG} \cup \text{ECDG}^0_{0.4}$) to 48.3 ($\text{CIG} \cup_{[1,4,1]} \text{ECDG}^0_{0.4}$). The overall trend of a slight improvement cannot be retrieved on project level. The individual values are varying too heavily: some increase while others decrease. But since the best average value increases in three of the four cases and stays constant in the other case, it is unlikely that this trend is a random result. Thus, the results suggest that integrating the structural and evolutionary graphs with the weighted union operation while emphasizing the twice occurring dependencies improves the clustering results as expected.

Comparing the weighted to the simple union experiment, the best clustering quality increases from 57.4 ($\text{SCDG} \cup \text{ECDG}^0_{0.8}$) to 58.5 ($\text{SCDG} \cup_{[1,4,1]} \text{ECDG}^1_{0.4}$). This underlines on the one hand once more that "it is possible to improve current software clustering algorithms that use structural information by integrating evolutionary data" (Hypothesis 2), and states on the other hand that twice occurring structural and evolutionary dependencies are more reliable for software clustering than other dependencies.

3.4.3 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-study implements such a weight optimization by comparing the clustering qualities of combined graphs created using different weighted union operations.

Due to runtime issues the weights can only be varied in five steps resulting in $5^3 = 125$ different weight setups. Moreover, as the number of combinations should not be increased further, only one structural as well as one evolutionary graph are considered: As the SCDG tends to be the best structural graph for software clustering, it is used to represent the structural graphs. As the $\text{ECDG}^1_{0.4}$ produces the best results in combination with the SCDG in the weighted union experiment, it is used to represent the evolutionary graphs. These two dependency graphs are united by the weighted union operation in 125 different setups. Additionally, the number of repeated *Bunch* clustering runs has to be decreased from 50, as used in all previous experiments, to 10. Hence, the results are more imprecise.

The search space is spanned by the set of weights $\{0, 1, 2, 4, 8\}$ for each weight parameter. To allow more extreme setups, the weights are elements of the exponential series of 2. The value of 0 is included to allow to ignore one or more of the three subsets: for example, $\cup_{[0,1,0]}$ is equivalent to a simple graph intersection operation. Although a systematically variation of the weights produces some redundancies (e.g., $\cup_{[1,1,1]}$ is equivalent to $\cup_{[2,2,2]}$), these redundant graphs are not omitted because they help to estimate the precision of the results.

Table 3.7 documents the parameter optimization experiment by a selection of the 15 best clustering results with respect to the average *MoJoFM* measure. The first and most important conclusion from the results is that the clustering quality cannot be improved in comparison to the weighted union $\cup_{[1,4,1]}$. In this experiment the results are, however, much more imprecise than before because of the lower number of repetitive runs: The average clustering quality for the combination with $\cup_{[1,4,1]}$ was 58.5 in the previous, more precise experiment (Section 3.4.2) but is now 59.5 and in the equivalent case of $\cup_{[2,8,2]}$ 59.2 respectively.

TABLE 3.7: Best *MoJoFM* clustering qualities based on the combined SCDG and $\text{ECDG}^1_{0.4}$ using the weighted union operation in different setups. The already examined weight setups are highlighted gray.

	Azureus	*JEdit*	*JFreeChart*	*JFtp*	*JUnit*	*Tomcat*	Average
SCDG	$48.8_{\pm 3.1}$	$68.1_{\pm 1.4}$	$43.4_{\pm 1.2}$	$53.5_{\pm 0.7}$	$58.5_{\pm 0.5}$	$54.4_{\pm 0.5}$	54.4
$\text{SCDG}\cup_{[1,4,1]}\text{ECDG}^1_{0.4}$	$53.6_{\pm 0.4}$	$70.9_{\pm 0.2}$	$56.0_{\pm 0.3}$	$62.7_{\pm 0.5}$	$57.4_{\pm 0.4}$	$56.3_{\pm 0.2}$	59.5
$\text{SCDG}\cup_{[1,8,2]}\text{ECDG}^1_{0.4}$	$53.7_{\pm 0.3}$	$71.4_{\pm 0.2}$	$57.0_{\pm 0.4}$	$60.6_{\pm 0.5}$	$58.5_{\pm 0.2}$	$55.5_{\pm 0.4}$	59.4
$\text{SCDG}\cup_{[2,8,2]}\text{ECDG}^1_{0.4}$	$53.3_{\pm 0.2}$	$70.9_{\pm 0.3}$	$56.0_{\pm 0.2}$	$62.8_{\pm 0.5}$	$56.4_{\pm 0.3}$	$55.6_{\pm 0.3}$	59.2
$\text{SCDG}\cup_{[1,8,1]}\text{ECDG}^1_{0.4}$	$53.6_{\pm 0.1}$	$69.6_{\pm 1.0}$	$56.3_{\pm 0.2}$	$62.5_{\pm 0.5}$	$57.8_{\pm 0.3}$	$53.8_{\pm 3.4}$	58.9
$\text{SCDG}\cup_{[2,4,1]}\text{ECDG}^1_{0.4}$	$54.2_{\pm 0.4}$	$70.0_{\pm 0.3}$	$52.8_{\pm 0.2}$	$63.0_{\pm 0.4}$	$58.5_{\pm 0.3}$	$54.7_{\pm 0.4}$	58.8
$\text{SCDG}\cup_{[2,4,2]}\text{ECDG}^1_{0.4}$	$53.3_{\pm 0.3}$	$69.6_{\pm 0.5}$	$56.4_{\pm 0.2}$	$59.0_{\pm 1.5}$	$57.3_{\pm 0.2}$	$57.0_{\pm 0.3}$	58.8
$\text{SCDG}\cup_{[4,8,4]}\text{ECDG}^1_{0.4}$	$53.9_{\pm 0.4}$	$69.9_{\pm 0.3}$	$55.5_{\pm 0.6}$	$58.0_{\pm 1.6}$	$56.8_{\pm 0.3}$	$56.3_{\pm 0.6}$	58.4
$\text{SCDG}\cup_{[1,1,1]}\text{ECDG}^1_{0.4}$	$52.6_{\pm 0.3}$	$70.3_{\pm 0.3}$	$56.7_{\pm 0.2}$	$54.1_{\pm 0.6}$	$58.7_{\pm 0.3}$	$56.9_{\pm 0.5}$	58.2
$\text{SCDG}\cup_{[2,2,1]}\text{ECDG}^1_{0.4}$	$51.8_{\pm 1.6}$	$69.6_{\pm 0.2}$	$52.7_{\pm 0.3}$	$61.7_{\pm 0.5}$	$58.3_{\pm 0.4}$	$55.0_{\pm 0.5}$	58.2
$\text{SCDG}\cup_{[8,8,4]}\text{ECDG}^1_{0.4}$	$53.5_{\pm 0.4}$	$69.8_{\pm 0.3}$	$51.9_{\pm 1.4}$	$61.0_{\pm 0.6}$	$58.4_{\pm 0.4}$	$54.8_{\pm 0.6}$	58.2
$\text{SCDG}\cup_{[1,2,1]}\text{ECDG}^1_{0.4}$	$53.5_{\pm 0.4}$	$70.2_{\pm 0.2}$	$51.3_{\pm 4.7}$	$59.0_{\pm 1.3}$	$57.1_{\pm 0.3}$	$57.3_{\pm 0.2}$	58.1
$\text{SCDG}\cup_{[2,8,4]}\text{ECDG}^1_{0.4}$	$53.3_{\pm 0.4}$	$71.0_{\pm 0.3}$	$56.8_{\pm 0.2}$	$54.2_{\pm 0.9}$	$57.8_{\pm 0.4}$	$55.4_{\pm 0.4}$	58.1
$\text{SCDG}\cup_{[2,1,2]}\text{ECDG}^1_{0.4}$	$51.2_{\pm 0.2}$	$70.7_{\pm 0.2}$	$56.5_{\pm 0.2}$	$53.0_{\pm 0.6}$	$59.2_{\pm 0.2}$	$57.3_{\pm 0.3}$	58.0
$\text{SCDG}\cup_{[4,4,2]}\text{ECDG}^1_{0.4}$	$54.0_{\pm 0.3}$	$69.9_{\pm 0.2}$	$50.2_{\pm 3.1}$	$60.7_{\pm 0.5}$	$58.7_{\pm 0.3}$	$54.5_{\pm 0.6}$	58.0
$\text{SCDG}\cup_{[4,8,2]}\text{ECDG}^1_{0.4}$	$51.1_{\pm 3.7}$	$70.3_{\pm 0.3}$	$52.8_{\pm 0.1}$	$61.3_{\pm 0.5}$	$58.0_{\pm 0.3}$	$54.7_{\pm 0.4}$	58.0

Despite the increased uncertainty, some general trends can be observed:

- Only in one of the 15 best cases listed in Table 3.7, a structural-only or evolutionary-only weight is larger than the mid-weight ($\text{SCDG} \cup_{[2,1,2]} \text{ECDG}^1_{0.4}$). In all other cases the mid-weight, which is representing twice occurring dependencies, is at least one of the strongest weights. An additional analysis considering all 125 setups confirms this: It shows that the mean clustering quality of the group of setups where the mid-weight belongs to the strongest weights is 57.9 whereas the rest only reaches 56.7. This result underlines again that twice occurring dependencies are more important for software clustering than the dependencies that are only included in one of the original graphs.

- Ignoring one of the dependency groups seems not to produce good clustering results because no combination with a weight of 0 for any group is represented in the list of the best combinations.
- The importance of the structural-only dependencies corresponds approximately to the importance of the evolutionary-only dependencies. The list of the 15 best results shows a minor preference of the structural dependencies: The weight for structural-only dependencies is higher than for the evolutionary-only ones in five cases and lower in only two cases. But averaging all setups with a higher structural than evolutionary weight produces only a clustering quality of 56.7 while the group with a higher evolutionary than structural weight produces slightly better qualities of 57.2.

From the list of the 15 best weight settings, only two ($\cup_{[1,1,1]}$ and $\cup_{[1,4,1]}$) have been already assessed in the previously used more precise experiment setup. But although the graph $\mathrm{SCDG} \cup_{[1,4,1]} \mathrm{ECDG}^1_{0.4}$ is top-ranked, a better weight setting may exist that coincidentally performs not as well in the optimization due to the low measuring precision. It is worth a try to assess another weighted union operation with the previously used exact experiment setup. Best candidate is obviously the weighted union operation $\cup_{[1,8,2]}$, which is the second best weight combination.

Table 3.8 presents the results for this experiment. Actually the clustering qualities increase slightly compared to the equivalent experiment based $\cup_{[1,4,1]}$ (Table 3.6): The best combined clustering quality rises

- for the CAG from 47.4 ($\mathrm{CAG} \cup_{[1,4,1]} \mathrm{ECDG}^0_{0.8}$) to 47.6 ($\mathrm{CAG} \cup_{[1,8,2]} \mathrm{ECDG}^0_{0.4}$),
- for the CUG from 57.2 ($\mathrm{CUG} \cup_{[1,4,1]} \mathrm{ECDG}^0_{0.4}$) to 57.5 ($\mathrm{CUG} \cup_{[1,8,2]} \mathrm{ECDG}^0_{0.8}$),
- for the SCDG from 58.5 ($\mathrm{SCDG} \cup_{[1,4,1]} \mathrm{ECDG}^1_{0.4}$) to 58.8 ($\mathrm{SCDG} \cup_{[1,8,2]} \mathrm{ECDG}^1_{0.4}$).

Only the clustering quality for the CIG decreases from 48.3 ($\mathrm{CIG} \cup_{[1,4,1]} \mathrm{ECDG}^0_{0.4}$) to 47.8 ($\mathrm{CIG} \cup_{[1,8,2]} \mathrm{ECDG}^0_{0.4}$). These marginal differences, however, may be a coincidental result.

Table 3.8: *MoJoFM* clustering quality based on combined structural and evolutionary dependency graphs using the weighted union operation $\cup_{[1,8,2]}$. The best average results are highlighted gray for each block.

	Azureus	*JEdit*	*JFreeChart*	*JFtp*	*JUnit*	*Tomcat*	Average
CIG	$30.2_{\pm 0.2}$	$42.0_{\pm 0.5}$	$35.4_{\pm 0.3}$	$53.4_{\pm 0.4}$	$58.0_{\pm 0.1}$	$32.8_{\pm 0.1}$	41.9
CAG	$28.6_{\pm 0.3}$	$47.1_{\pm 0.7}$	$14.3_{\pm 0.0}$	$45.7_{\pm 0.3}$	$55.9_{\pm 0.0}$	$35.8_{\pm 0.0}$	37.9
CUG	$52.4_{\pm 0.4}$	$63.5_{\pm 1.7}$	$40.6_{\pm 0.6}$	$49.4_{\pm 0.0}$	$58.8_{\pm 0.2}$	$54.5_{\pm 0.5}$	53.2
SCDG	$49.9_{\pm 1.0}$	$65.5_{\pm 1.6}$	$44.2_{\pm 0.4}$	$54.0_{\pm 0.3}$	$58.4_{\pm 0.2}$	$54.0_{\pm 0.5}$	54.3
$\mathrm{ECDG}^0_{0.0}$	$31.7_{\pm 0.1}$	$40.4_{\pm 0.1}$	$40.7_{\pm 0.2}$	$41.5_{\pm 0.2}$	$60.5_{\pm 0.1}$	$11.1_{\pm 0.0}$	37.7
$\mathrm{ECDG}^0_{0.4}$	$35.6_{\pm 0.1}$	$47.0_{\pm 0.2}$	$43.2_{\pm 0.1}$	$42.3_{\pm 0.4}$	$61.9_{\pm 0.1}$	$11.3_{\pm 0.0}$	40.2
$\mathrm{ECDG}^0_{0.8}$	$32.6_{\pm 0.3}$	$45.9_{\pm 0.1}$	$39.8_{\pm 0.1}$	$42.7_{\pm 0.3}$	$60.0_{\pm 0.0}$	$11.4_{\pm 0.0}$	38.7
$\mathrm{ECDG}^1_{0.4}$	$31.3_{\pm 0.2}$	$39.6_{\pm 0.4}$	$28.2_{\pm 0.1}$	$35.2_{\pm 0.0}$	$57.2_{\pm 0.0}$	$7.5_{\pm 0.0}$	33.2
$\mathrm{ECDG}^1_{0.8}$	$27.0_{\pm 0.0}$	$36.9_{\pm 0.1}$	$20.9_{\pm 0.0}$	$35.4_{\pm 0.2}$	$55.6_{\pm 0.0}$	$6.8_{\pm 0.0}$	30.4
$\mathrm{CIG} \cup_{[1,8,2]} \mathrm{ECDG}^0_{0.0}$	$34.1_{\pm 0.4}$	$42.6_{\pm 1.4}$	$51.0_{\pm 0.1}$	$43.6_{\pm 0.2}$	$59.7_{\pm 0.0}$	$34.5_{\pm 0.1}$	44.2
$\mathrm{CIG} \cup_{[1,8,2]} \mathrm{ECDG}^0_{0.4}$	$37.5_{\pm 0.6}$	$50.8_{\pm 1.0}$	$53.5_{\pm 0.1}$	$50.0_{\pm 0.3}$	$60.9_{\pm 0.1}$	$34.3_{\pm 0.4}$	47.8
$\mathrm{CIG} \cup_{[1,8,2]} \mathrm{ECDG}^0_{0.8}$	$36.1_{\pm 0.1}$	$53.6_{\pm 0.2}$	$54.9_{\pm 0.1}$	$46.7_{\pm 0.3}$	$60.0_{\pm 0.0}$	$34.8_{\pm 0.0}$	47.7
$\mathrm{CIG} \cup_{[1,8,2]} \mathrm{ECDG}^1_{0.4}$	$35.8_{\pm 0.2}$	$51.4_{\pm 0.6}$	$45.9_{\pm 0.7}$	$53.2_{\pm 0.3}$	$59.7_{\pm 0.0}$	$34.6_{\pm 0.1}$	46.8
$\mathrm{CIG} \cup_{[1,8,2]} \mathrm{ECDG}^1_{0.8}$	$34.7_{\pm 0.1}$	$54.1_{\pm 0.2}$	$45.1_{\pm 0.2}$	$55.4_{\pm 0.3}$	$59.2_{\pm 0.0}$	$33.9_{\pm 0.1}$	47.1
$\mathrm{CAG} \cup_{[1,8,2]} \mathrm{ECDG}^0_{0.0}$	$36.3_{\pm 0.1}$	$52.4_{\pm 0.9}$	$43.0_{\pm 0.1}$	$44.6_{\pm 0.3}$	$60.8_{\pm 0.1}$	$36.6_{\pm 0.1}$	45.6
$\mathrm{CAG} \cup_{[1,8,2]} \mathrm{ECDG}^0_{0.4}$	$37.0_{\pm 0.1}$	$58.4_{\pm 0.3}$	$44.5_{\pm 0.4}$	$47.4_{\pm 0.2}$	$61.9_{\pm 0.1}$	$36.1_{\pm 0.1}$	47.6
$\mathrm{CAG} \cup_{[1,8,2]} \mathrm{ECDG}^0_{0.8}$	$35.0_{\pm 0.0}$	$59.0_{\pm 0.4}$	$41.2_{\pm 0.4}$	$46.2_{\pm 0.3}$	$61.3_{\pm 0.1}$	$36.0_{\pm 0.4}$	46.5
$\mathrm{CAG} \cup_{[1,8,2]} \mathrm{ECDG}^1_{0.4}$	$34.3_{\pm 0.0}$	$54.2_{\pm 0.5}$	$32.3_{\pm 0.1}$	$45.7_{\pm 0.1}$	$58.6_{\pm 0.0}$	$35.5_{\pm 0.3}$	43.5
$\mathrm{CAG} \cup_{[1,8,2]} \mathrm{ECDG}^1_{0.8}$	$32.5_{\pm 0.0}$	$56.5_{\pm 0.2}$	$27.4_{\pm 0.1}$	$47.1_{\pm 0.3}$	$57.0_{\pm 0.0}$	$35.8_{\pm 0.0}$	42.7
$\mathrm{CUG} \cup_{[1,8,2]} \mathrm{ECDG}^0_{0.0}$	$54.7_{\pm 0.9}$	$63.8_{\pm 1.1}$	$51.1_{\pm 0.8}$	$48.0_{\pm 0.4}$	$58.2_{\pm 0.2}$	$52.0_{\pm 0.1}$	54.6
$\mathrm{CUG} \cup_{[1,8,2]} \mathrm{ECDG}^0_{0.4}$	$54.4_{\pm 1.1}$	$67.5_{\pm 1.1}$	$53.8_{\pm 0.9}$	$50.6_{\pm 0.3}$	$57.5_{\pm 0.1}$	$53.7_{\pm 0.7}$	56.2
$\mathrm{CUG} \cup_{[1,8,2]} \mathrm{ECDG}^0_{0.8}$	$52.5_{\pm 0.8}$	$68.3_{\pm 1.2}$	$52.9_{\pm 1.1}$	$55.2_{\pm 0.3}$	$58.7_{\pm 0.3}$	$57.2_{\pm 0.2}$	57.5
$\mathrm{CUG} \cup_{[1,8,2]} \mathrm{ECDG}^1_{0.4}$	$54.7_{\pm 0.5}$	$67.3_{\pm 1.0}$	$54.8_{\pm 0.3}$	$49.5_{\pm 0.2}$	$58.1_{\pm 0.2}$	$55.6_{\pm 0.6}$	56.6
$\mathrm{CUG} \cup_{[1,8,2]} \mathrm{ECDG}^1_{0.8}$	$52.6_{\pm 0.5}$	$68.7_{\pm 0.5}$	$49.3_{\pm 0.8}$	$45.7_{\pm 0.3}$	$57.5_{\pm 0.2}$	$57.1_{\pm 0.1}$	55.2
$\mathrm{SCDG} \cup_{[1,8,2]} \mathrm{ECDG}^0_{0.0}$	$53.7_{\pm 1.1}$	$63.9_{\pm 1.5}$	$52.9_{\pm 1.0}$	$50.2_{\pm 0.3}$	$58.9_{\pm 0.1}$	$52.5_{\pm 0.1}$	55.3
$\mathrm{SCDG} \cup_{[1,8,2]} \mathrm{ECDG}^0_{0.4}$	$53.2_{\pm 0.9}$	$69.1_{\pm 0.9}$	$55.7_{\pm 0.2}$	$54.5_{\pm 0.3}$	$58.3_{\pm 0.1}$	$54.9_{\pm 0.1}$	57.6
$\mathrm{SCDG} \cup_{[1,8,2]} \mathrm{ECDG}^0_{0.8}$	$50.9_{\pm 0.5}$	$69.3_{\pm 1.7}$	$55.4_{\pm 0.7}$	$51.7_{\pm 0.3}$	$58.6_{\pm 0.1}$	$56.9_{\pm 0.1}$	57.1
$\mathrm{SCDG} \cup_{[1,8,2]} \mathrm{ECDG}^1_{0.4}$	$54.0_{\pm 0.1}$	$68.7_{\pm 1.2}$	$56.6_{\pm 0.3}$	$59.7_{\pm 0.3}$	$58.2_{\pm 0.1}$	$55.6_{\pm 0.2}$	58.8
$\mathrm{SCDG} \cup_{[1,8,2]} \mathrm{ECDG}^1_{0.8}$	$52.0_{\pm 0.2}$	$66.9_{\pm 1.4}$	$51.9_{\pm 0.8}$	$53.4_{\pm 0.3}$	$57.8_{\pm 0.1}$	$56.3_{\pm 0.5}$	56.4

Chapter 4

Study on Architecture Improvements

After focusing on architecture recovery in the previous chapter, this chapter evaluates the usefulness of software clustering to propose improvements for software architectures. It studies the relevance of different data sources on this task. An automatically created software decomposition usually differs from the factual decomposition defined through the architecture of the system. The previous study tried to minimize this difference because it aims to recover the architecture. In contrast, the current study will use this difference to propose architecture improvements: The elements of the difference are classes that are placed into a different cluster by the automatic clustering than by the factual decomposition. This misplacement may be the result from either a weakness in the clustering algorithm—an accidental difference—or a design flaw in the actual decomposition—an intended difference.

This change of perspective—from considering the difference between the automatically generated and the actual decomposition as harmful to considering them as useful—is no contradiction: As discussed in Section 2.5.1, it has to be assumed that the factual decomposition derived from the project package structure is not perfect, and thus, the difference between both decompositions might include design flaws (i.e., intended differences). By minimizing the decomposition difference in a single project both kinds of differences, accidental and intended, are equally minimized. But by examining a set of projects only systematic differences are minimized because they occur in every project. Assumed that design flaws are not likely to occur systematically in different projects, intended differences are not systematic. Thus, the optimization of a clustering algorithm as conducted in the previous study reduces the accidental differences while it keeps the intended differences.

The clustering decompositions are often compared to a reference decomposition to evaluate a software clustering approach (Section 2.5.1). This evaluation method focuses on architecture recovery and cannot answer the question whether the clustering technique is able to detect design flaws in the architecture. As described in Section 1.3.2, Vanya et al. [64] use software clustering based on evolutionary data to identify such design flaws by comparing the clustering decomposition to the factual architecture. To evaluate this approach, experts assessed the design flaws manually and confirmed most of them to be valuable.

The present study addresses the same application of software clustering as the study by Vanya et. al. but is conducted under different constraints: It is supposed to

- compare evolutionary, structural, and combined data sources,
- work automatically as far as possible (thus, do not employ experts).

To meet these requirements, I developed a novel evaluation method. This method uses the version archives to identify transactions where the software architecture has changed. Assuming that, at least in the average case, the developer changed the architecture intentionally, such a transaction is likely to improve the architecture. A clustering setup is good if the clustering decomposition based on the data before the architecture change is able to predict the subsequent change.

4.1 Identifying Architectural Improvements

During development, the architecture of a software system is sometimes partly redesigned. This redesign process results in a changed package structure of the system. But a changed package structure is just a necessary, not a sufficient condition for an architecture redesign: For example, a new class would also change the structure. Hence, all changes that extend features of the program, solve errors, or alter the semantic of the program in any other way are irrelevant for the present application. It is sufficient to focus on refactorings, which are changes that do not alter the semantic of the program by definition [27].

There exist lots of different refactoring types, for example, *ExtractMethod*, *RemoveParameter*, or *HideMethod* refactorings (an exhaustive list can be found in [27]). For the purpose of detecting architecture changes, only refactorings that change the package structure are interesting, refactorings on finer-grained artifacts are not relevant. This reduces the set of important refactoring types to *MoveClass* and *MoveInterface*. A moved package is considered as a set of multiple *MoveClass* and *MoveInterface* refactorings. Thus, as a heuristic, the transactions that change the system architecture are identified by the occurrence of several such move refactorings in one transaction.

The present study uses an approach presented by Weißgerber and Diehl [68, 67] to automatically detect such refactorings from version archives. At first, their technique scans the version archive and parses the sources on a fine-grained level (it uses the same library as the present work does to obtain the evolutionary dependencies, see Section 2.3.2). Then, it identifies refactoring candidates by analyzing the changes of class and method signatures—every refactoring type has its characteristic signature change pattern. The candidates are ranked and filtered with the help of a code similarity measurement. The filtered candidates finally form the set of detected refactorings. Beside many other refactoring types, this technique is able to find the required *MoveClass* and *MoveInterface* refactorings.

The parameters of the detection process are chosen in a conservative way that prefers reliability over completeness of the detected refactorings. These two terms, reliability and completeness, can also be expressed as the precision and recall of the detected refactorings compared to the actually applied ones. Based on the evaluation by Weißgerber ([67], Chapter 4), the filter parameters are set to a very reliable filter called *b1c75* with an estimated precision value of 1.0 (i.e., 100% of the detected refactorings are correct) and an estimated recall value of 0.87 (i.e., 87% of the actual refactorings are detected) for class-artifacts refactorings such as *MoveClass* and *MoveInterface*.

This refactoring detection technique was applied to the whole set of sample software projects. Every transaction with at least two *MoveClass* or *MoveInterface* refactorings (excluding very early move refactorings originating from the default package) is considered as a relevant architecture change. Table 4.1 presents the resulting numbers of relevant transactions: Although thousands of transactions are included in the examined time frame, a total of only twelve transactions were found. These transactions mainly pool at the *Azureus* and *JEdit* project, each with five transactions. Hence, for this second study only these two projects are worth considering.

Table 4.1: Number of transactions with at least two detected *MoveClass* and *MoveInterface* refactorings.

	Azureus	*JEdit*	*JFreeChart*	*JFtp*	*JUnit*	*Tomcat*	Sum
# Relevant transactions	5	5	1	0	1	0	12

4.2 Predictive Capability of Software Decompositions

Assessing the usefulness of the different data sources for predicting architecture changes, a different evaluation method than in the previous study has to be applied. One of the main difference is that the clustering algorithm is not allowed to work on the latest checked-out version but actually on the version just before the considered refactoring happened. Thus, the experimenter has to check-out every predecessor version of a detected relevant architecture change and has to compile it manually.

While the preprocessing and the clustering works exactly as before, an unmodified assessment method is not useful: By now, the architecture change has to be somehow considered to estimate the predictive capability of the clustering decomposition. This is realized by comparing the similarity of the clustering decomposition and the software architecture before the architecture change with the similarity after the particular architecture change. If the similarity increases, the clustering decomposition predicts the the architecture change at least partly because the clustering decomposition is more similar to the improved architecture.

The *MoJoFM* measure is not suitable to compute these similarities because it would normalize the metric value based on two different reference decompositions. The distance measure based on the number of necessary *Move* and *Join* operations (mno function, Definition 2.13), however, provides an adequate tool. The mno difference is considered as a predictive capability measure of the clustering decompositions.

Definition 4.1 (Predictive Capability)**.** *Consider a clustering decomposition A, a reference decomposition B according to the software architecture before the change, and a reference decomposition B' according to the software architecture after the change. The* ***Predictive Capability*** pc *is defined as a function*

$$\begin{array}{rcl} \mathrm{pc} : \mathfrak{P}_{C(S)} \times \mathfrak{P}_{C(S)} \times \mathfrak{P}_{C(S)} & \to & \mathbb{Z} \\ (A, B, B') & \mapsto & \mathrm{pc}(A, B, B') := \mathrm{mno}(A, B) - \mathrm{mno}(A, B') \end{array}$$

Hence, a positive pc value indicates a higher similarity to the changed architecture, i.e., a good *Predictive Capability*. A pc value of zero represents situations when the clustering decomposition is equally similar to both reference decompositions. And finally, negative values are also possible: the clustering decomposition is more similar to the architecture before the change.

Concurrently to the *MoveClass* and *MoveInterface* refactorings, non-refactoring changes might also alter the architecture in the same transaction. Thus, it is more reliable to create a virtual reference decomposition B' that only applies the detected changes to decomposition B than to use the real architecture after the change, which might be influenced by additional changes. Nevertheless, decomposition B is generated like a normal reference decomposition in the previous experiments.

4.3 Experimental Results

According to the previous description, the experiments of the present second study assess the *Predictive Capability* of the automatically generated clustering decomposition. Because of the low number of relevant architecture change transactions in other projects, only *Azureus* and *JEdit* are considered. Besides the individual structural and evolutionary graphs, the clustering uses the best combined graph setup derived from the previous study, which is based on the $\cup_{[1,8,2]}$ weighted union operation. The precision of the *Predictive Capability* results is increased by 50 repetitive clustering runs.

Table 4.2 and Table 4.3 present the results for *Azureus* and *JEdit*. Each column represents a certain version identified by the given transaction ID of the architecture change transaction (only the structural and evolutionary data before that transaction are taken into account for the clustering). The number of moved classes in the respective transactions give a roughly estimated maximum bound for the absolute *Predictive Capability*: x moved classes can only result in a maximum mno difference of x.

It is obvious that the resulting *Predictive Capability* values are much more volatile than the *MoJoFM* values in the previous experiments. For instance, the average values in the last block (combinations of SCDG and ECDG) vary from -0.3 up to 0.4 for *Azureus* and even from -1.2 up to -0.1 for *JEdit*. In contrast, these combined graphs produce very similar clustering results in the previous study. Moreover, comparing the results of different transactions does not directly show any similar behaviour of the values. The reason of this volatility is probably the fact that the moved classes only affect minimal parts of the software system: Coincidentally, one of the dependency graphs might include information about this part, other graphs might not. Since the experiments only consider five transactions, the sample size is probably just too small. Nevertheless, the number of transactions cannot be increased because all possible transactions are already included. Thus, for the assessed projects, the results at hand cannot be sharpened easily. They are partly random and have to be interpreted cautiously.

Among the structural graphs, CUG and SCDG seems to perform best as they produce the best results for both projects (0.2 and -0.4 respectively). Among the evolutionary

Table 4.2: *Predictive Capability* of the clustering results based on combined structural and evolutionary dependency graphs using the weighted union operation $\cup_{[1,8,2]}$ for *Azureus* at relevant architecture changes.

Azureus	Txn 611	Txn 774	Txn 792	Txn 936	Txn 962	Average
# moved classes	3	10	4	4	2	
CIG	$0.0_{\pm 0.0}$	$0.0_{\pm 0.1}$	$-1.0_{\pm 0.0}$	$1.9_{\pm 0.0}$	$-2.0_{\pm 0.0}$	-0.2
CAG	$0.0_{\pm 0.0}$	$-2.5_{\pm 0.1}$	$0.0_{\pm 0.0}$	$0.0_{\pm 0.0}$	$-2.0_{\pm 0.0}$	-0.9
CUG	$0.6_{\pm 0.1}$	$-1.4_{\pm 0.1}$	$-0.2_{\pm 0.1}$	$1.3_{\pm 0.1}$	$0.7_{\pm 0.1}$	0.2
SCDG	$0.4_{\pm 0.1}$	$-0.9_{\pm 0.1}$	$0.0_{\pm 0.0}$	$1.3_{\pm 0.1}$	$0.3_{\pm 0.1}$	0.2
ECDG$^0_{0.0}$	$-3.0_{\pm 0.0}$	$0.5_{\pm 0.2}$	$-0.9_{\pm 0.0}$	$-1.1_{\pm 0.0}$	$1.0_{\pm 0.0}$	-0.7
ECDG$^0_{0.4}$	$-3.0_{\pm 0.0}$	$1.0_{\pm 0.0}$	$-1.0_{\pm 0.0}$	$-1.0_{\pm 0.0}$	$1.0_{\pm 0.0}$	-0.6
ECDG$^0_{0.8}$	$-2.0_{\pm 0.0}$	$0.0_{\pm 0.0}$	$-1.0_{\pm 0.0}$	$-1.3_{\pm 0.1}$	$1.0_{\pm 0.0}$	-0.7
ECDG$^1_{0.4}$	$-1.0_{\pm 0.0}$	$-2.0_{\pm 0.0}$	$0.0_{\pm 0.0}$	$0.0_{\pm 0.0}$	$-2.0_{\pm 0.0}$	-1.0
ECDG$^1_{0.8}$	$-1.0_{\pm 0.0}$	$-2.0_{\pm 0.0}$	$0.0_{\pm 0.0}$	$0.0_{\pm 0.0}$	$-2.0_{\pm 0.0}$	-1.0
CIG $\cup_{[1,8,2]}$ ECDG$^0_{0.0}$	$0.0_{\pm 0.0}$	$0.3_{\pm 0.1}$	$-1.2_{\pm 0.1}$	$1.9_{\pm 0.1}$	$1.0_{\pm 0.0}$	0.4
CIG $\cup_{[1,8,2]}$ ECDG$^0_{0.4}$	$0.0_{\pm 0.0}$	$0.7_{\pm 0.1}$	$-1.5_{\pm 0.1}$	$1.9_{\pm 0.1}$	$1.0_{\pm 0.0}$	0.4
CIG $\cup_{[1,8,2]}$ ECDG$^0_{0.8}$	$0.0_{\pm 0.0}$	$-0.3_{\pm 0.3}$	$-1.6_{\pm 0.1}$	$1.9_{\pm 0.0}$	$1.0_{\pm 0.0}$	0.2
CIG $\cup_{[1,8,2]}$ ECDG$^1_{0.4}$	$-0.1_{\pm 0.0}$	$0.0_{\pm 0.2}$	$-1.0_{\pm 0.0}$	$1.9_{\pm 0.0}$	$-2.0_{\pm 0.0}$	-0.2
CIG $\cup_{[1,8,2]}$ ECDG$^1_{0.8}$	$0.0_{\pm 0.0}$	$-0.3_{\pm 0.2}$	$-1.0_{\pm 0.0}$	$1.7_{\pm 0.1}$	$-2.0_{\pm 0.0}$	-0.3
CAG $\cup_{[1,8,2]}$ ECDG$^0_{0.0}$	$0.0_{\pm 0.0}$	$-0.2_{\pm 0.1}$	$-1.8_{\pm 0.1}$	$0.0_{\pm 0.0}$	$0.9_{\pm 0.1}$	-0.2
CAG $\cup_{[1,8,2]}$ ECDG$^0_{0.4}$	$0.0_{\pm 0.0}$	$0.0_{\pm 0.0}$	$-1.4_{\pm 0.1}$	$0.0_{\pm 0.0}$	$1.0_{\pm 0.0}$	-0.1
CAG $\cup_{[1,8,2]}$ ECDG$^0_{0.8}$	$0.0_{\pm 0.0}$	$-0.3_{\pm 0.1}$	$-1.4_{\pm 0.1}$	$0.0_{\pm 0.0}$	$0.9_{\pm 0.1}$	-0.2
CAG $\cup_{[1,8,2]}$ ECDG$^1_{0.4}$	$0.0_{\pm 0.0}$	$-1.9_{\pm 0.0}$	$-0.1_{\pm 0.1}$	$0.0_{\pm 0.0}$	$-2.0_{\pm 0.0}$	-0.8
CAG $\cup_{[1,8,2]}$ ECDG$^1_{0.8}$	$0.0_{\pm 0.0}$	$-2.0_{\pm 0.0}$	$0.0_{\pm 0.0}$	$0.0_{\pm 0.0}$	$-2.0_{\pm 0.0}$	-0.8
CUG $\cup_{[1,8,2]}$ ECDG$^0_{0.0}$	$0.1_{\pm 0.0}$	$0.0_{\pm 0.0}$	$-3.3_{\pm 0.1}$	$1.2_{\pm 0.1}$	$1.0_{\pm 0.0}$	-0.2
CUG $\cup_{[1,8,2]}$ ECDG$^0_{0.4}$	$0.4_{\pm 0.1}$	$0.0_{\pm 0.0}$	$-2.7_{\pm 0.1}$	$1.2_{\pm 0.1}$	$1.0_{\pm 0.0}$	0.0
CUG $\cup_{[1,8,2]}$ ECDG$^0_{0.8}$	$0.0_{\pm 0.0}$	$-0.6_{\pm 0.1}$	$-2.9_{\pm 0.1}$	$1.3_{\pm 0.1}$	$1.0_{\pm 0.0}$	-0.2
CUG $\cup_{[1,8,2]}$ ECDG$^1_{0.4}$	$0.0_{\pm 0.0}$	$-1.0_{\pm 0.0}$	$-0.1_{\pm 0.1}$	$1.1_{\pm 0.1}$	$0.9_{\pm 0.1}$	0.2
CUG $\cup_{[1,8,2]}$ ECDG$^1_{0.8}$	$0.0_{\pm 0.0}$	$-1.1_{\pm 0.0}$	$-0.2_{\pm 0.1}$	$1.2_{\pm 0.1}$	$0.9_{\pm 0.1}$	0.1
SCDG $\cup_{[1,8,2]}$ ECDG$^0_{0.0}$	$0.2_{\pm 0.1}$	$0.0_{\pm 0.0}$	$-2.0_{\pm 0.0}$	$2.0_{\pm 0.0}$	$1.0_{\pm 0.0}$	0.2
SCDG $\cup_{[1,8,2]}$ ECDG$^0_{0.4}$	$0.3_{\pm 0.1}$	$0.3_{\pm 0.1}$	$-1.4_{\pm 0.1}$	$2.0_{\pm 0.0}$	$1.0_{\pm 0.0}$	0.4
SCDG $\cup_{[1,8,2]}$ ECDG$^0_{0.8}$	$0.1_{\pm 0.0}$	$-0.3_{\pm 0.1}$	$-2.3_{\pm 0.1}$	$1.1_{\pm 0.0}$	$-0.2_{\pm 0.2}$	-0.3
SCDG $\cup_{[1,8,2]}$ ECDG$^1_{0.4}$	$0.0_{\pm 0.0}$	$-0.2_{\pm 0.1}$	$0.0_{\pm 0.0}$	$1.6_{\pm 0.1}$	$0.6_{\pm 0.1}$	0.4
SCDG $\cup_{[1,8,2]}$ ECDG$^1_{0.8}$	$0.0_{\pm 0.0}$	$-0.3_{\pm 0.1}$	$-0.1_{\pm 0.1}$	$1.4_{\pm 0.1}$	$-0.1_{\pm 0.2}$	0.2

TABLE 4.3: *Predictive Capability* of the clustering results based on combined structural and evolutionary dependency graphs using the weighted union operation $\cup_{[1,8,2]}$ for *JEdit* at relevant architecture changes.

JEdit	Txn 4630	Txn 4678	Txn 5114	Txn 5225	Txn 7129	Average
# moved classes	4	8	3	2	6	
CIG	0.0±0.0	-3.5±0.1	0.0±0.0	0.0±0.0	1.0±0.0	-0.5
CAG	-3.1±0.2	-3.8±0.3	2.0±0.0	0.4±0.1	0.7±0.1	-0.8
CUG	-2.6±0.3	-2.1±0.2	0.5±0.4	1.5±0.2	0.9±0.1	-0.4
SCDG	-2.8±0.3	-1.4±0.1	-0.4±0.4	1.7±0.1	0.9±0.1	-0.4
ECDG$^0_{0.0}$	0.0±0.0	-3.2±0.2	-0.6±0.1	0.0±0.0	1.0±0.0	-0.6
ECDG$^0_{0.4}$	0.0±0.0	-3.9±0.1	-2.0±0.0	0.0±0.0	1.0±0.0	-1.0
ECDG$^0_{0.8}$	0.0±0.0	-3.5±0.1	-2.0±0.0	0.0±0.0	1.0±0.0	-0.9
ECDG$^1_{0.4}$	0.0±0.0	-1.0±0.0	-2.0±0.0	0.0±0.0	0.2±0.1	-0.6
ECDG$^1_{0.8}$	0.0±0.0	-0.8±0.1	-2.0±0.0	0.0±0.0	0.6±0.1	-0.4
CIG $\cup_{[1,8,2]}$ ECDG$^0_{0.0}$	0.0±0.0	-3.3±0.1	-0.5±0.1	0.0±0.0	0.9±0.1	-0.6
CIG $\cup_{[1,8,2]}$ ECDG$^0_{0.4}$	0.0±0.0	-2.8±0.2	-1.7±0.1	0.0±0.0	0.9±0.0	-0.7
CIG $\cup_{[1,8,2]}$ ECDG$^0_{0.8}$	0.0±0.0	-2.1±0.1	-1.7±0.1	0.0±0.0	1.0±0.0	-0.6
CIG $\cup_{[1,8,2]}$ ECDG$^1_{0.4}$	0.0±0.0	-1.2±0.1	-1.9±0.1	0.0±0.0	1.0±0.0	-0.4
CIG $\cup_{[1,8,2]}$ ECDG$^1_{0.8}$	0.0±0.0	-2.0±0.1	-2.0±0.0	0.0±0.0	0.9±0.0	-0.6
CAG $\cup_{[1,8,2]}$ ECDG$^0_{0.0}$	0.0±0.0	-1.1±0.2	0.6±0.1	0.8±0.1	0.8±0.1	0.2
CAG $\cup_{[1,8,2]}$ ECDG$^0_{0.4}$	0.0±0.0	-4.1±0.4	0.0±0.0	-0.1±0.1	0.9±0.1	-0.7
CAG $\cup_{[1,8,2]}$ ECDG$^0_{0.8}$	0.0±0.0	-3.4±0.1	-1.0±0.0	0.0±0.1	0.1±0.1	-0.8
CAG $\cup_{[1,8,2]}$ ECDG$^1_{0.4}$	0.0±0.0	-0.2±0.1	-1.0±0.1	0.8±0.1	0.0±0.0	-0.1
CAG $\cup_{[1,8,2]}$ ECDG$^1_{0.8}$	0.0±0.0	-3.4±0.4	1.5±0.3	0.3±0.1	0.2±0.1	-0.3
CUG $\cup_{[1,8,2]}$ ECDG$^0_{0.0}$	-0.7±0.1	-2.8±0.2	-1.5±0.2	-1.2±0.1	0.9±0.0	-1.1
CUG $\cup_{[1,8,2]}$ ECDG$^0_{0.4}$	-0.7±0.1	-1.4±0.1	-2.0±0.0	0.7±0.1	0.3±0.1	-0.6
CUG $\cup_{[1,8,2]}$ ECDG$^0_{0.8}$	-0.8±0.1	-3.3±0.1	-2.9±0.1	1.4±0.1	-1.3±0.1	-1.4
CUG $\cup_{[1,8,2]}$ ECDG$^1_{0.4}$	-2.6±0.3	-2.2±0.2	1.5±0.2	1.1±0.2	0.9±0.0	-0.3
CUG $\cup_{[1,8,2]}$ ECDG$^1_{0.8}$	-2.9±0.3	-1.4±0.1	-2.9±0.1	1.2±0.2	1.0±0.0	-1.0
SCDG $\cup_{[1,8,2]}$ ECDG$^0_{0.0}$	-0.6±0.1	-3.4±0.3	-1.9±0.2	-1.1±0.1	1.0±0.0	-1.2
SCDG $\cup_{[1,8,2]}$ ECDG$^0_{0.4}$	-0.8±0.1	-1.7±0.1	-1.9±0.1	1.0±0.1	0.6±0.1	-0.6
SCDG $\cup_{[1,8,2]}$ ECDG$^0_{0.8}$	-0.8±0.1	-2.9±0.2	-2.8±0.1	1.2±0.2	0.6±0.1	-1.0
SCDG $\cup_{[1,8,2]}$ ECDG$^1_{0.4}$	-2.9±0.3	-1.8±0.1	1.8±0.1	1.4±0.1	1.0±0.0	-0.1
SCDG $\cup_{[1,8,2]}$ ECDG$^1_{0.8}$	-2.8±0.3	-2.2±0.1	-3.0±0.0	1.2±0.2	1.0±0.0	-1.2

graphs, one cannot determine a best candidate because the results are too inconsistent. In general, the structural graphs seem to perform slightly better than the evolutionary graphs: The average *Predictive Capability* intervals are for *Azureus* $[-0.9, 0.2]$ (structural) and $[-1.0, -0.6]$ (evolutionary), and for *JEdit* $[-0.8, -0.4]$ (structural) and $[-1.0, -0.4]$ (evolutionary). The results for the combined graphs in both projects are again too inconsistent to be assessed.

Nevertheless, the weak findings of this second study match with the findings from the study on architecture recovery:

- CUG and SCDG are the most reliable structural graphs.
- Structural graphs produce slightly better clustering qualities than evolutionary graphs.

Hence, the clustering results might show similar behaviour for both applications in general. Although the latest results are not able to really strengthen the evidence of the assumed hypotheses further, they neither contradict them in any way. If the experiment included more projects, this would probably lead to more and stronger findings.

Chapter 5

Discussion

The experiments of this thesis provide diverse insights in the clustering capabilities of structural and evolutionary data. This chapter summarizes and discusses these findings and furthermore gives an outlook on possible future research.

5.1 Contributions

A first important contribution of this work is the development of a reliable experimental environment for software clustering. The *Software Clustering Analysis Suite* (*SCAS*) bundles a flexible graph based data model, a successful software clustering algorithm, and a dependable evaluation method with controlling and analyzing features. This environment design enables to conduct all experiments of this study. Moreover, it provides a good infrastructure for further studies because it can be easily extended with other data sources, clustering algorithms, or evaluation methods.

The study on architecture recovery, which is a major application of software clustering, yields some strong findings. The first clustering experiment compares single structural and evolutionary graphs directly. It shows that the evolutionary data produces competitive clustering results in many cases: The best average evolutionary clustering quality is reached with a filtering by confidence ($\text{ECDG}^{0}_{0.4}$). The quality of this clustering is about as high as the quality of the structural graphs based on inheritance and aggregation information (CIG and CAG). Only taking usage dependencies into account (CUG and SCDG) results in better clustering qualities. Nevertheless, for two of the six projects the best evolutionary qualities even reach the best structural quality. For one project, though, the clustering based on evolutionary clustering does not work reasonably well. A closer look at the difference between structural and evolutionary dependency graphs shows that structural graphs are more dense while evolutionary graphs provide a better dependency quality. Due to the sparse dependencies, clustering with only evolutionary data is supposed to be more fragile than clustering with only structural data.

Moreover, adding evolutionary data to the structural data improves structural clustering quality clearly. Already a data integration with the non-weighted union operation shows this effect. Using a weighted union operation that emphasizes twice occurring dependencies (structural and evolutionary dependencies) enhances the quality even more. A weight optimization finally is able to improve the result slightly again. Note that a quality enhancement is reached in every project, even if the evolutionary dependency data is weak.

The weight optimization experiment allows to compare the following three groups of edges: structural-only, twice occurring, and evolutionary-only edges. Every group contributes helpful information for clustering because excluding one of them leads to inferior clustering results in the experiment. Among the three groups of edges twice occurring edges are most important while structural-only edges tend to be as important as the evolutionary-only edges.

All in all, the results confirm Hypothesis 1 and Hypothesis 2 for the application of architecture recovery and the examined projects: Evolutionary data is valuable for software clustering and is able to improve current software clustering approaches based on structural data.

The second study addresses the software clustering application of architecture improvement. This work introduces a novel evaluation method to assess to what extent a clustering algorithm is able to predict architecture changes. Since the method is based on refactoring detection and needs not to employ experts, it is totally automatable. But the experiments based on that method show that only very few transactions among the thousands examined ones include relevant architecture changes. This small data base is not enough to detect similarly strong findings as in the first study. Nevertheless, the results of the second study tend to confirm the results of the first study: CUG and SCDG are the best structural data sources. Furthermore, the two graphs appear to yield better results than the single evolutionary graphs. It cannot be said to what extent a combination of data sources is successful. Finally, the two hypotheses can neither be confirmed nor discarded in the application of architecture improvement because of statistical uncertainty.

5.2 Threats to Validity

The results of a study are at first only valid for the examined subjects, here, for the sample software projects in a certain version. If the findings cannot be applied to a larger context, their relevance is limited. The current study is based on six different software projects that cover a wide range of application types, however restricted to *Java* projects with less than 1000 classes. Thus, the results of the present study can be considered an indication for general *Java* projects and still a weak indication for other software projects.

The clustering algorithm *Bunch* is a very important part of the experiment but is not the examined subject. Improving the clustering results with *Bunch* does not automatically mean that this is also possible for other clustering algorithms in the same way. In fact, the whole data model and the data source integration mechanism has to be adapted when using another algorithm. But the example of *Bunch* indirectly showed the increased data quality for software clustering integrating structural and evolutionary data sources. Probably other algorithms are also able to use this data quality improvement to produce better clustering results.

Evaluations in the domain of software clustering are always difficult because no simple natural quality criterion for software decompositions exists. In Section 2.5 the author discussed internal and external assessment methods and chose an appropriate state-of-the-art evaluation method based on the *MoJoFM* metric and the factual architecture as the reference decomposition. The main problems of this technique are that the factual architecture might not be a good reference decomposition and that *MoJoFM* might not be an appropriate quality measure. At least, if these problems occurred only in one or two of the projects, the number of six projects would help to restrict their impact.

The present study focuses on two main use cases of software clustering: architecture recovery and architecture improvement. The employed evaluation technique, however, only provides reliable results for the use case of architecture recovery. Thus, the results are at first only valid for this application. Nevertheless, it is plausible that other use cases of software clustering would also profit from the applied approach of integrating structural and 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. 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).

5.3 Related Work

This thesis adopts and extends recent approaches on integrating software clustering and software evolution (Section 1.3). The basic idea of using evolutionary information for clustering software artifacts was already introduced by Ball et al. [9]. They model the data as a dependency graph similar to the one used in the current study. Further related approaches use clustering on evolutionary data to detect flaws in the software architecture [64] or to improve a visualization of version archives [65, 66]. But yet no study has compared the quality of an evolution based software clustering approach to a state-of-the-art structural based software clustering approach like it is done in the current study.

Andritsos and Tzerpos [3, 4] came up with the idea of enriching structural data with non-structural data for software clustering. Among the non-structural data, they use file ownership and file timestamp—two information sources that are evolutionary to some extent. Furthermore, Wierda et al. [72] combine two version of the same software project to improve the software clustering. This is also a strategy that uses some kind of evolutionary information. Nevertheless, none of these integration approaches use the rich evolutionary data source of change transactions. The present work is the first one that successfully integrates this data source into a structural based software clustering approach.

Since single sample projects might depart significantly from the average case, it is important to examine enough projects. While many studies only employ one or two sample projects (e.g., [3, 42, 43, 49, 56, 62, 64, 71]), only a few use five or more projects [48, 50, 74]. The present study uses six sample software projects. Furthermore, a broad spectrum of projects in terms of software type and size is necessary to guarantee a certain generalizability of the results. Obviously, such a spectrum can only be provided by the more extensive studies like the one at hand. A minor restriction of the current study, however, is the project size limit of 1000 classes (*JEdit* is the largest sample with 840 classes). But only few other studies cluster much more software artifacts (e.g., up to 3900 [11] or 3266 [74]). Most studies vary a certain part of their experimental design depending on what they aim to assess: the data source [4, 72], the clustering algorithm [50, 70], the evaluation method [48, 71], or several parts [5, 46]. A set of different evaluation methods is also employed to increase the reliability of the results [46, 50]. Since the present study aims to compare different data sources, the kind of data source is varied as the independent variable. Moreover, the data sources are assessed in two use cases of software clustering.

Summing up, the presented study is

- the first that systematically compares structural and evolutionary data sources for software clustering,
- the first that integrates rich evolutionary data into a state-of-the-art clustering approach based on structural data, and
- even one of the most extensive studies in the domain of software clustering in general.

5.4 Future Research

To improve the generalizability of the results, it would be desirable to extend the study to more projects as well as to other programming languages and paradigms. Additionally, it would be also interesting to investigate closed source projects. Furthermore, the efficiency of the preprocessing and clustering has to be increased, for example, by avoiding the manual compilation of the source code or by using a faster clustering algorithm.

It would be very interesting to repeat the study with a different clustering algorithm and an adapted data model, especially with the feature-based *LIMBO* algorithm as discussed in Section 2.4.2. The results can be compared to the present study and the question might be answered which of the approaches is more suitable to integrate structural and evolutionary data.

But also the potential of the current approach might not be exhaustively used. Varying the diverse parameters systematically or trying other data integration methods might improve the clustering quality even more. A first step in this direction is the optimization of the weight parameters of the weighted union operation as performed in Section 3.4.3. Other parameters that would be promising to optimize are, for example, the threshold values of the ECDG, further (weighted) union operations among structural and evolutionary graphs, or advanced edge weighting schemes for the dependency graphs. Such optimizations, however, need a faster clustering algorithm and/or a more elaborate optimization strategy.

Finally, other evolution based or non-structural data types that might be able to also improve structural clustering results exist: for example, the file ownership information (as already used in [3, 4]), class dependencies through bugs or through mail communication. Integrating all these data types into one data structure might provide a very rich data base for software clustering. As the number of parameters increases with every additional data source, a good parameter optimization strategy would be necessary again.

5.5 Conclusion

The results of the present study show that evolutionary information from version archives is often able to provide a sufficient data set for a high quality software clustering and, moreover, is capable to improve a clustering result based on extensive structural information. Thus, the study clearly identifies software evolution as a valuable data source for software clustering.

Furthermore, the data-centered experiments demonstrate how important the choice and processing of data sources in the domain of software clustering is. The present results suggest that researchers will be able to improve the quality of software clustering considerably by further extending the integration of structural and non-structural data sources.

Bibliography

[1] S. K. Abd-El-Hafiz. Identifying objects in procedural programs using clustering neural networks. *Automated Software Engineering*, 7(3):239–261, July 2000.

[2] B. Andreopoulos, A. An, V. Tzerpos, and X. Wang. Clustering large software systems at multiple layers. *Information and Software Technology*, 49(3):244–254, March 2007.

[3] P. Andritsos and V. Tzerpos. Software clustering based on information loss minimization. In *WCRE '03: Proceedings of the 10th Working Conference on Reverse Engineering*, pages 334–344, Washington, DC, USA, 2003. IEEE Computer Society.

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

[5] N. Anquetil, C. Fourrier, and T. C. Lethbridge. Experiments with clustering as a software remodularization method. In *WCRE '99: Proceedings of the Sixth Working Conference on Reverse Engineering*, pages 235–255, Washington, DC, USA, 1999. IEEE Computer Society.

[6] N. Anquetil and T. Lethbridge. File clustering using naming conventions for legacy systems. In *CASCON '97: Proceedings of the 1997 Conference of the Centre for Advanced Studies on Collaborative Research.* IBM Press, 1997.

[7] Apache Tomcat, `http://tomcat.apache.org`.

[8] Azureus, `http://azureus.sourceforge.net`.

[9] T. Ball, J.-M. Kim, A. A. Porter, and H. P. Siy. If your version control system could talk ... In *ICSE '97 Workshop on Process Modelling and Empirical Studies of Software Engineering.* ACM Press, May 1997.

[10] M. Bauer and M. Trifu. Architecture-aware adaptive clustering of OO systems. In *CSMR '04: Proceedings of the Eighth Euromicro Working Conference on Software Maintenance and Reengineering*, Washington, DC, USA, 2004. IEEE Computer Society.

[11] D. Beyer and A. Noack. Clustering software artifacts based on frequent common changes. In *IWPC '05: Proceedings of the 13th International Workshop on Program Comprehension*, pages 259–268. IEEE Computer Society, 2005.

[12] D. Beyer and A. Noack. Mining co-change clusters from version repositories. Technical report, École Polytechnique Fédérale de Lausanne, January 2005.

[13] J. Bortz. *Statistik für Sozialwissenschaftler*. Springer, 5th edition, 1999.

[14] I. T. Bowman and R. C. Holt. Software architecture recovery using Conway's law. In *CASCON '98: Proceedings of the 1998 Conference of the Centre for Advanced Studies on Collaborative Research*. IBM Press, 1998.

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

[16] Bunch, `http://serg.cs.drexel.edu/tiki-index.php?page=Bunch`.

[17] M. Burch, F. Beck, and S. Diehl. Timeline Trees: Visualizing sequences of transactions in information hierarchies. In *AVI '08: Proceedings of 9th International Working Conference on Advanced Visual Interfaces*, pages 75–82, Naples, Italy, 2008.

[18] M. Burch and S. Diehl. TimeRadarTrees: Visualizing dynamic compound digraphs. *Computer Graphics Forum*, 27(3):823–830, 2008.

[19] M. Burch, S. Diehl, and P. Weißgerber. Visual data mining in software archives. In *SoftVis '05: Proceedings of the 2005 ACM Symposium on Software Visualization*, pages 37–46, New York, NY, USA, 2005. ACM Press.

[20] Y. Chiricota, F. Jourdan, and G. Melançon. Software components capture using graph clustering. In *IWPC '03: Proceedings of the 11th IEEE International Workshop on Program Comprehension*, Washington, DC, USA, 2003. IEEE Computer Society.

[21] C. Collberg, S. Kobourov, J. Nagra, J. Pitts, and K. Wampler. A system for graph-based visualization of the evolution of software. In *SoftVis '03: Proceedings of the 2003 ACM Symposium on Software Visualization*, pages 77–86, New York, NY, USA, 2003. ACM Press.

[22] DependencyFinder, `http://depfind.sourceforge.net`.

[23] S. Diehl. *Software Visualization: Visualizing the Structure, Behaviour, and Evolution of Software*. Springer, May 2007.

[24] D. Doval, S. Mancoridis, and B. S. Mitchell. Automatic clustering of software systems using a genetic algorithm. In *STEP '99: Proceedings of the 1999 International Conference on Software Tools and Engineering Practice*, pages 73–81. IEEE Computer Society, 1999.

[25] L. H. Etzkorn and C. G. Davis. Automatically identifying reusable OO legacy code. *Computer*, 30(10):66–71, October 1997.

[26] D. Flanagan. *Java In A Nutshell.* O'Reilly Media, Inc., 5th edition, March 2005.

[27] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley Professional, June 1999.

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

[29] H. Gall, M. Jazayeri, and C. Riva. Visualizing software release histories: The use of color and third dimension. In *ICSM '99: Proceedings of the IEEE International Conference on Software Maintenance*, pages 99–108, Los Alamitos, CA, USA, 1999. IEEE Computer Society.

[30] J. Gargiulo and S. Mancoridis. Gadget: A tool for extracting the dynamic structure of Java programs. In *SEKE '01: Proceedings of the Thirteenth International Conference on Software Engineering and Knowledge Engineering*, pages 244–251, 2001.

[31] GraphML, `http://graphml.graphdrawing.org`.

[32] JEdit, `http://www.jedit.org`.

[33] JFreeChart, `http://www.jfree.org/jfreechart`.

[34] JFtp, `http://j-ftp.sourceforge.net`.

[35] JGraphT, `http://www.jgrapht.org/`.

[36] JUnit, `http://www.junit.org`.

[37] S. Kim, T. Zimmermann, J. E. Whitehead, and A. Zeller. Predicting faults from cached history. In *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*, pages 489–498, Washington, DC, USA, 2007. IEEE Computer Society.

[38] R. Koschke and T. Eisenbarth. A framework for experimental evaluation of clustering techniques. In *IWPC '00: Proceedings of the 8th International Workshop on Program Comprehension*, pages 201–210, Washington, DC, USA, 2000. IEEE Computer Society.

[39] A. Kuhn, S. Ducasse, and T. Girba. Enriching reverse engineering with semantic clustering. In *WCRE '05: Proceedings of the 12th Working Conference on Reverse Engineering*, pages 133–142, Washington, DC, USA, 2005. IEEE Computer Society.

[40] S. Li and L. Tahvildari. JComp: A reuse-driven componentization framework for Java applications. In *ICPC '06: Proceedings of the 14th IEEE International Conference on Program Comprehension*, pages 264–267, Washington, DC, USA, 2006. IEEE Computer Society.

[41] Y. S. Maarek, D. M. Berry, and G. E. Kaiser. An information retrieval approach for automatically constructing software libraries. *IEEE Transactions on Software Engineering*, 17(8):800–813, August 1991.

[42] J. I. Maletic and A. Marcus. Supporting program comprehension using semantic and structural information. In *ICSE '01: Proceedings of the 23rd International Conference on Software Engineering*, pages 103–112, Washington, DC, USA, 2001. IEEE Computer Society.

[43] S. Mancoridis, B. S. Mitchell, Y. Chen, and E. R. Gansner. 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*, pages 50–59, Washington, DC, USA, 1999. IEEE Computer Society.

[44] S. Mancoridis, B. S. Mitchell, C. Rorres, Y. Chen, and E. R. Gansner. Using automatic clustering to produce high-level system organizations of source code. In *IWPC '98: Proceedings of the 6th International Workshop on Program Comprehension*, pages 45–52, Washington, DC, USA, 1998. IEEE Computer Society.

[45] O. Maqbool and H. A. Babri. The Weighted Combined Algorithm: A linkage algorithm for software clustering. In *CSMR '04: Proceedings of the 8th Euromicro Working Conference on Software Maintenance and Reengineering*, pages 15+, Washington, DC, USA, 2004. IEEE Computer Society.

[46] O. Maqbool and H. A. Babri. Hierarchical clustering for software architecture recovery. *IEEE Transactions on Software Engineering*, 33(11):759–780, 2007.

[47] B. S. Mitchell. *A Heuristic Approach to Solving the Software Clustering Problem.* PhD thesis, Drexel University, March 2002.

[48] B. S. Mitchell and S. Mancoridis. Comparing the decompositions produced by software clustering algorithms using similarity measurements. In *ICSM '01: Proceedings of the 17th IEEE International Conference on Software Maintenance*, pages 744–753, 2001.

[49] B. S. Mitchell and S. Mancoridis. CRAFT: A framework for evaluating software clustering results in the absence of benchmark decompositions. In *WCRE '01: Proceedings of the Eighth Working Conference on Reverse Engineering*, pages 93–102, Washington, DC, USA, 2001. IEEE Computer Society.

[50] B. S. Mitchell and S. Mancoridis. On the evaluation of the Bunch search-based software modularization algorithm. *Soft Computing*, 12(1):77–93, August 2007.

[51] MoJo, `http://www.cse.yorku.ca/~bil/downloads/`.

[52] H. A. Müller and J. S. Uhl. Composing subsystem structures using (k,2)-partite graphs. In *Proceedings of the Conference on Software Maintenance 1990*, pages 12–19, 1990.

[53] D. Rayside, S. Reuss, E. Hedges, and K. Kontogiannis. The effect of call graph construction algorithms for object-oriented programs on automatic clustering. In *IWPC '00: Proceedings of the 8th International Workshop on Program Comprehension*, pages 191–200, Washington, DC, USA, 2000. IEEE Computer Society.

[54] M. Saeed, O. Maqbool, H. A. Babri, S. Z. Hassan, and S. M. Sarwar. Software clustering techniques and the use of Combined Algorithm. In *CSMR '03: Proceedings of the Seventh European Conference on Software Maintenance and Reengineering*, pages 301–306, Washington, DC, USA, 2003. IEEE Computer Society.

[55] R. W. Schwanke. An intelligent tool for re-engineering software modularity. In *ICSE '91: Proceedings of the 13th International Conference on Software Engineering*, pages 83–92, Los Alamitos, CA, USA, 1991. IEEE Computer Society.

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

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

[58] M. Siff and T. Reps. Identifying modules via concept analysis. *IEEE Transactions on Software Engineering*, 25(6):749–768, November 1999.

[59] W. P. Stevens, G. J. Myers, and L. L. Constantine. Structured design. *IBM Systems Journal*, 13(2):115–139, 1974.

[60] V. Tzerpos. *Comprehension-Driven Software Clustering.* PhD thesis, University of Toronto, 2001.

[61] V. Tzerpos and R. C. Holt. MoJo: A distance metric for software clusterings. In *WCRE '99: Proceedings of the Sixth Working Conference on Reverse Engineering*, pages 187–193, Washington, DC, USA, 1999. IEEE Computer Society.

[62] V. Tzerpos and R. C. Holt. ACDC: An algorithm for comprehension-driven clustering. In *WCRE '00: Proceedings of the Seventh Working Conference on Reverse Engineering*, pages 258–267. IEEE Computer Society, 2000.

[63] A. van Deursen and T. Kuipers. Identifying objects using cluster and concept analysis. In *ICSE '99: Proceedings of the 21st International Conference on Software Engineering*, pages 246–255, Los Alamitos, CA, USA, 1999. IEEE Computer Society.

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

[65] L. Voinea and A. Telea. CVSgrab: Mining the history of large software projects. In *EuroVis '06: Joint Eurographics - IEEE VGTC Symposium on Visualization*, pages 187–194. Eurographics Association, May 2006.

[66] L. Voinea and A. Telea. Multiscale and multivariate visualizations of software evolution. In *SoftVis '06: Proceedings of the 2006 ACM Symposium on Software Visualization*, pages 115–124, New York, NY, USA, 2006. ACM.

[67] P. Weißgerber. *Automatic Refactoring Detection in Version Archives.* PhD thesis, Univerisity of Trier, 2009.

[68] P. Weißgerber and S. Diehl. Identifying refactorings from source-code changes. In *ASE '06: Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, pages 231–240, Washington, DC, USA, 2006. IEEE Computer Society.

[69] Z. Wen and V. Tzerpos. An optimal algorithm for MoJo distance. In *IWPC '03: Proceedings of the 11th IEEE International Workshop on Program Comprehension*, pages 227–235, Washington, DC, USA, 2003. IEEE Computer Society.

[70] Z. Wen and V. Tzerpos. An effectiveness measure for software clustering algorithms. In *IWPC '04: Proceedings of the 12th International Workshop on Program Comprehension*, pages 194–203. IEEE Computer Society, 2004.

[71] Z. Wen and V. Tzerpos. Evaluating similarity measures for software decompositions. In *ICSM '04: Proceedings of the 20th IEEE International Conference on Software Maintenance*, pages 368–377, Washington, DC, USA, 2004. IEEE Computer Society.

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

[73] T. A. Wiggerts. Using clustering algorithms in legacy systems remodularization. In *WCRE '97: Proceedings of the Fourth Working Conference on Reverse Engineering*, Washington, DC, USA, 1997. IEEE Computer Society.

[74] J. Wu, A. E. Hassan, and R. C. Holt. Comparison of clustering algorithms in the context of software evolution. In *ICSM '05: Proceedings of the 21st IEEE International Conference on Software Maintenance*, pages 525–535, Washington, DC, USA, 2005. IEEE Computer Society.

[75] C. Xiao and V. Tzerpos. Software clustering based on dynamic dependencies. In *CSMR '05: Proceedings of the Ninth European Conference on Software Maintenance and Reengineering*, pages 124–133, Washington, DC, USA, 2005. IEEE Computer Society.

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

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

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

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